

USENIX

conference

.....
proceedings

Proceedings of the 6th USENIX Conference on File and Storage Technologies

San Jose, CA USA February 26–29, 2008

6th USENIX Conference on File and Storage Technologies

*San Jose, CA, USA
February 26–29, 2008*

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGOPS,
IEEE Mass Storage Systems Technical
Committee (MSSTC), and IEEE TCOS

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • <http://www.usenix.org>

The price is \$35 for members and \$45 for nonmembers.
Outside the U.S.A. and Canada, please add \$15 per copy for postage (via air printed matter).

Thanks to Our Reception Sponsor



Thanks to Our Silver Sponsors



Thanks to Our Sponsors



Thanks to Our Media Sponsors

<i>ACM Queue</i>	<i>Linux Pro Magazine</i>
ITtoolbox	LXer.com
<i>Linux Journal</i>	StorageNetworking.org
<i>Linux+DVD</i>	The Register

© 2008 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN-13: 978-1-931971-56-0
ISBN-10: 1-931971-56-0

USENIX Association

**Proceedings of the
6th USENIX Conference on
File and Storage Technologies
(FAST '08)**

**February 26–29, 2008
San Jose, CA, USA**

Conference Organizers

Program Co-Chairs

Mary Baker, *Hewlett-Packard Labs*
Erik Riedel, *Seagate Research*

Program Committee

Andrea C. Arpaci-Dusseau, *University of Wisconsin, Madison*
Randal Burns, *Johns Hopkins University*
Howard Gobioff, *Google*
Christos Karamanolis, *VMware*
Kim Keeton, *Hewlett-Packard Labs*
Geoff Kuenning, *Harvey Mudd College*
Darrell Long, *University of California, Santa Cruz*
Petros Maniatis, *Intel Research Berkeley*
Robert Morris, *Massachusetts Institute of Technology*
Brian Noble, *University of Michigan*
Alma Riska, *Seagate Research*
Antony Rowstron, *Microsoft Research, UK*
Jiri Schindler, *NetApp*
Margo Seltzer, *Harvard University*
Doug Terry, *Microsoft*
Theodore Ts'o, *IBM*
Andrew Warfield, *University of British Columbia*
Ric Wheeler, *EMC*
Theodore Wong, *IBM Research*
Erez Zadok, *Stony Brook University*

Tutorial Chair

Richard Golding, *IBM Almaden Research Center*

Work-in-Progress Reports and Posters Chair

Geoff Kuenning, *Harvey Mudd College*

Steering Committee

Andrea C. Arpaci-Dusseau, *University of Wisconsin, Madison*
Remzi H. Arpaci-Dusseau, *University of Wisconsin, Madison*
Jeff Chase, *Duke University*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas*
Peter Honeyman, *CITI, University of Michigan, Ann Arbor*
Merritt Jones, *MITRE Corporation*
Darrell Long, *University of California, Santa Cruz*
Jai Menon, *IBM Research*
Margo Seltzer, *Harvard University*
Chandu Thekkah, *Microsoft Research*
John Wilkes, *Hewlett-Packard Labs*
Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

Salimah Addetia	Stephen Fridella	Ethan Miller	Keith Smith
Mike Adl-El-Malek	Kevin Greenan	Reddy Narasimha	Sai Susarla
Matt Amdur	John Griffin	Dushyanth Narayanan	Doug Thain
Ahmed Amer	Sudhanva Gurumurthi	Shankar Pasupathy	Niranjan Thirumale
Alexander "Sasha" Ames	Jim Hafner	Adam Pennington	Mithuna S. Thottethodi
Ralph Becker-Szendy	James Hendricks	Xiao Qin	Niraj Tolia
Ross Biro	Val Henson	Tim Rausch	Murali Vilayannur
David Black	Sami Iren	Brandon Salmon	Rosie Wacha
Liz Borowsky	Nikhil Jagtiani	Dan Scales	Steven Whitehouse
James Bottomley	Hong Jiang	Bianca Schroeder	Qin Xin
Angela Brown	Mahmut Kandemir	Thomas Schwarz	Bennet Yee
Jose Brustoloni	Arkady Kanevsky	Rusty Sears	Qi Zhang
Fabian Bustamante	Andy Klosterman	Dave Seekins	Yifeng Zhu
Steve Byan	Andrew Leung	Mehul Shah	
John Chandy	Wei-keng Liao	Minglong Shao	
Patrick Eaton	Kiran Madnani	Kai Shen	
Daniel Ellard	Mallik Mahalingam	Evgenia Smirni	

6th USENIX Conference on File and Storage Technologies
February 26–29, 2008
San Jose, CA, USA

Index of Authors	v
Message from the Program Co-Chairs	vi

Wednesday, February 27

Distributed Storage

Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage	1
<i>Mark W. Storer, Kevin M. Greenan, and Ethan L. Miller, University of California, Santa Cruz; Kaladhar Voruganti, Network Appliance</i>	
Scalable Performance of the Panasas Parallel File System	17
<i>Brent Welch, Marc Unangst, and Zainul Abbasi, Panasas, Inc.; Garth Gibson, Panasas, Inc., and Carnegie Mellon University; Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou, Panasas, Inc.</i>	
TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions	35
<i>Michael Demmer, Bowei Du, and Eric Brewer, University of California, Berkeley</i>	

You Cache, I Cache...

On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better Than Demotions	49
<i>Binny S. Gill, IBM Almaden Research Center</i>	
AWOL: An Adaptive Write Optimizations Layer	67
<i>Alexandros Batsakis and Randal Burns, Johns Hopkins University; Arkady Kanevsky, James Lentini, and Thomas Talpey, Network Appliance™ Inc.</i>	
TaP: Table-based Prefetching for Storage Caches	81
<i>Mingju Li, Elizabeth Varki, and Swapnil Bhatia, University of New Hampshire; Arif Merchant, Hewlett-Packard Labs</i>	

Thursday, February 28

Failures and Loss

The RAID-6 Liberation Codes	97
<i>James S. Plank, University of Tennessee</i>	
Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics	111
<i>Weihsang Jiang, Chongfeng Hu, and Yuanyuan Zhou, University of Illinois at Urbana-Champaign; Arkady Kanevsky, Network Appliance, Inc.</i>	
Parity Lost and Parity Regained	127
<i>Andrew Krioukov and Lakshmi N. Bairavasundaram, University of Wisconsin, Madison; Garth R. Goodson, Kiran Srinivasan, and Randy Thelen, Network Appliance, Inc.; Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	

CPUs, Compilers, and Packets, Oh My!

Enhancing Storage System Availability on Multi-Core Architectures with Recovery-Conscious Scheduling143
Sangeetha Seshadri, Georgia Institute of Technology; Lawrence Chiu, Cornel Constantinescu, Subashini Balachandran, and Clem Dickey, IBM Almaden Research Center; Ling Liu, Georgia Institute of Technology; Paul Muench, IBM Almaden Research Center

Improving I/O Performance of Applications through Compiler-Directed Code Restructuring159
Mahmut Kandemir and Seung Woo Son, Pennsylvania State University; Mustafa Karakoy, Imperial College

Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems175
Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan, Carnegie Mellon University

Where Did We Go Wrong?

Portably Solving File TOCTTOU Races with Hardness Amplification189
Dan Tsafir, IBM T.J. Watson Research Center; Tomer Hertz, Microsoft Research; David Wagner, University of California, Berkeley; Dilma Da Silva, IBM T.J. Watson Research Center

EIO: Error Handling is Occasionally Correct207
Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit, University of Wisconsin, Madison

An Analysis of Data Corruption in the Storage Stack223
Lakshmi N. Bairavasundaram, University of Wisconsin, Madison; Garth R. Goodson, Network Appliance, Inc.; Bianca Schroeder, University of Toronto; Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

Friday, February 29

Buffers, Power, and Bottlenecks

BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage239
Hyojun Kim and Seongjun Ahn, Software Laboratory of Samsung Electronics, Korea

Write Off-Loading: Practical Power Management for Enterprise Storage253
Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron, Microsoft Research Ltd.

Avoiding the Disk Bottleneck in the Data Domain Deduplication File System269
Benjamin Zhu, Data Domain, Inc.; Kai Li, Data Domain, Inc., and Princeton University; Hugo Patterson, Data Domain, Inc.

Compliance and Provisioning

Towards Tamper-evident Storage on Patterned Media283
Pieter H. Hartel, Leon Abelman, and Mohammed G. Khatib, University of Twente, The Netherlands

SWEEPER: An Efficient Disaster Recovery Point Identification Mechanism297
Akshat Verma, IBM India Research; Kaladhar Voruganti, Network Appliance; Ramani Routray, IBM Almaden Research; Rohit Jain, Yahoo India

Using Utility to Provision Storage Systems313
John D. Strunk, Carnegie Mellon University; Eno Thereska, Microsoft Research, Cambridge UK; Christos Faloutsos and Gregory R. Ganger, Carnegie Mellon University

Index of Authors

Abbasi, Zainul	17	Li, Kai	269
Abelmann, Leon	283	Li, Mingju	81
Ahn, Seongjun	239	Liblit, Ben	207
Andersen, David G.	175	Liu, Ling	143
Arpaci-Dusseau, Andrea C.	127, 207, 223	Merchant, Arif	81
Arpaci-Dusseau, Remzi H.	127, 207, 223	Miller, Ethan L.	1
Bairavasundaram, Lakshmi N.	127, 223	Mueller, Brian	17
Balachandran, Subashini	143	Muench, Paul	143
Batsakis, Alexandros	67	Narayanan, Dushyanth.	253
Bhatia, Swapnil	81	Patterson, Hugo	269
Brewer, Eric	35	Phanishayee, Amar	175
Burns, Randal	67	Plank, James S.	97
Chiu, Lawrence	143	Routray, Ramani	297
Constantinescu, Cornel	143	Rowstron, Antony	253
Da Silva, Dilma	189	Rubio-González, Cindy	207
Demmer, Michael	35	Schroeder, Bianca	223
Dickey, Clem	143	Seshadri, Sangeetha	143
Donnelly, Austin	253	Seshan, Srinivasan	175
Du, Bowei	35	Small, Jason	17
Faloutsos, Christos.	313	Son, Seung Woo	159
Ganger, Gregory R.	175, 313	Srinivasan, Kiran	127
Gibson, Garth A.	17, 175	Storer, Mark W.	1
Gill, Binny S.	49	Strunk, John D.	313
Goodson, Garth R.	127, 223	Talpey, Thomas	67
Greenan, Kevin M.	1	Thelen, Randy	127
Gunawi, Haryadi S.	207	Thereska, Eno	313
Hartel, Pieter H.	283	Tsafrir, Dan	189
Hertz, Tomer	189	Unangst, Marc	17
Hu, Chongfeng.	111	Varki, Elizabeth	81
Jain, Rohit	297	Vasudevan, Vijay	175
Jiang, Weihang	111	Verma, Akshat	297
Kandemir, Mahmut	159	Voruganti, Kaladhar.	1, 297
Kanevsky, Arkady	67, 111	Wagner, David	189
Karakoy, Mustafa.	159	Welch, Brent	17
Khatib, Mohammed G.	283	Zelenka, Jim.	17
Kim, Hyojun	239	Zhou, Bin	17
Krevat, Elie	175	Zhou, Yuanyuan	111
Krioukov, Andrew	127	Zhu, Benjamin	269
Lentini, James	67		

Message from the Program Co-Chairs

We are delighted to welcome you to the 6th USENIX Conference on File and Storage Technologies. This year's conference promises to be very exciting, offering a diverse set of topics and an excellent balance across academic and industrial research efforts from around the world.

Given that this is the first time FAST has been scheduled on a yearly rather than an 18-month basis, we were very gratified to receive 94 high-quality submissions. The task of choosing the 21 papers presented in our technical program took a tremendous amount of time and dedication from our program committee and external reviewers, to whom we extend our sincerest thanks. Through an arduous review process, with over 370 reviews produced, the committee narrowed down our proposed selections to a few more than 50 papers before we met in person for two half-days in October 2007 to determine the program. At the end of this meeting, the committee's work was not done: all accepted papers were assigned program committee shepherds to help the authors get the most out of the reviewers' suggestions and to ensure the highest-quality proceedings. Our committee members must also choose the best paper awards and chair technical sessions at the conference. We were very impressed with the reliability, hard work, insight, and graciousness of our committee members. It has been a great pleasure to work with them! Finally, we would like to thank all of the authors who submitted work to the conference, regardless of whether we were able to include it in the proceedings.

We would also like to thank many other people who have helped make this conference possible. Program committee member Geoff Kuenning tripled his duties to perform also as chair of the work-in-progress and poster sessions. Last year's program chairs, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, and the FAST Steering Committee kindly answered our many questions and addressed all our concerns. Richard Golding put together an excellent set of tutorials. We are honored to have Cathy Marshall and Chandrakant Patel as keynote speakers. As always, the USENIX staff, in particular Ellie Young, Devon Shaw, Casey Henderson, Jane-Ellen Long, and Peter Collinson, has provided immediate and cheerful support throughout the process.

Welcome to FAST '08. We hope the conference is productive, exciting, and enjoyable for you.

Mary Baker, HP Labs
Erik Riedel, Seagate Research
FAST '08 Co-Chairs

Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage

Mark W. Storer Kevin M. Greenan Ethan L. Miller Kaladhar Voruganti
University of California, Santa Cruz *Network Appliance*

Abstract

As the world moves to digital storage for archival purposes, there is an increasing demand for reliable, low-power, cost-effective, easy-to-maintain storage that can still provide adequate performance for information retrieval and auditing purposes. Unfortunately, no current archival system adequately fulfills all of these requirements. Tape-based archival systems suffer from poor random access performance, which prevents the use of inter-media redundancy techniques and auditing, and requires the preservation of legacy hardware. Many disk-based systems are ill-suited for long-term storage because their high energy demands and management requirements make them cost-ineffective for archival purposes.

Our solution, Pergamum, is a distributed network of intelligent, disk-based, storage appliances that stores data reliably and energy-efficiently. While existing MAID systems keep disks idle to save energy, Pergamum adds NVRAM at each node to store data signatures, metadata, and other small items, allowing deferred writes, metadata requests and inter-disk data verification to be performed while the disk is powered off. Pergamum uses both intra-disk and inter-disk redundancy to guard against data loss, relying on hash tree-like structures of algebraic signatures to efficiently verify the correctness of stored data. If failures occur, Pergamum uses staggered rebuild to reduce peak energy usage while rebuilding large redundancy stripes. We show that our approach is comparable in both startup and ongoing costs to other archival technologies and provides very high reliability. An evaluation of our implementation of Pergamum shows that it provides adequate performance.

1 Introduction

Businesses and consumers are becoming increasingly conscious of the value of archival data. In the business

arena, data preservation is often mandated by law, and data mining has proven to be a boon in shaping business strategy. For individuals, archival storage is being called upon to preserve sentimental and historical artifacts such as photos, movies and personal documents. In both of these areas, archival systems must keep pace with a growing need for efficient, reliable, long-term storage.

Many storage systems designed for long-term data preservation rely on sequential-access technologies, such as tapes, that decouple media from its access hardware. While effective for back-up workloads (write-once, read-rarely, newer writes supersede old), such systems are poorly suited to archival workloads (write-once, read-maybe, new writes unrelated to old writes). With as many as 50–100 tapes per drive, a requirement to keep tapes running at full speed, and a linear media-access model, random-access performance with tape-media is relatively poor. This conspires against many archival storage operations — such as auditing, searching, consistency checking and inter-media reliability operations — that rely on relatively fast random-access performance. This is especially important in light of the preservation and retrieval demands of recent legislation [23, 30]. Further, many data retention policies include the notion of a limited lifetime, after which data is securely deleted; selective deletion is difficult and inefficient in linear media. Finally, the separation of media and access hardware introduces the need to preserve complex chains of hardware; reading an old tape requires a compatible reader, controller and software.

Recently, hard drives have dropped in price relative to tape, making them a potential alternative for archival storage [33]. The availability of high-performance, low-power CPUs [4] and inexpensive, high-speed networks have made it possible to produce a self-contained, network-attached storage device [16] with reasonable performance and low power utilization: as little as 500 mW when both the CPU and disk are idle. The use of disks instead of tapes means that heads are packaged

with media, removing the need for robotics and reducing physical movement and system complexity. Using standardized communication interfaces, such as TCP/IP over Ethernet, also helps simplify technology migration and long-term maintenance. By using randomly-accessible disks instead of linear tapes, systems can take advantage of inter-media redundancy schemes. Unfortunately, many existing disk-based systems incur high costs associated with power, cooling and administration because of design approaches that favor performance over energy efficiency. However, recent work on MAIDs (Massive Arrays of Idle Disks) has demonstrated that considerable energy-based cost savings can be realized while maintaining high levels of performance [10, 32, 45], though such systems often favor performance over even greater energy savings.

Our design differs from that of existing MAID systems, which still have centralized controllers. Instead, our system, Pergamum, takes an approach similar to that used in high-performance scalable storage systems [36, 46, 48], and is built from thousands of intelligent storage appliances connected by high-speed networks that cooperatively provide reliable, efficient, long-term storage. Each appliance, called a Pergamum *tome*, is composed of four hardware components: a commodity hard drive for persistent, large-capacity storage; on-board flash memory for persistent, low-latency, metadata storage; a low-power CPU; and a network port. Each appliance runs its own copy of the Pergamum software, allowing it to manage its own consistency checking, disk scrubbing and redundancy group responsibilities. Additionally, the CPU and extensible software layer enables disk-level processing, such as compression and virus checking. Finally, the use of standardized networking interfaces and protocols greatly reduces the problem of maintaining complex chains of dependent hardware.

Pergamum introduces several new techniques to disk-based archival storage. First, our system distributes control to the individual devices, rather than centralizing it, by including a low-power CPU and network interface on each disk; this approach reduces power consumption by eliminating the need for power-hungry servers and RAID controllers. Systems such as TickerTAIP [8] used distributed control in a RAID, but did not include reliability checking and power management. Second, Pergamum aggressively ensures data reliability using two forms of redundancy: intra-disk and inter-disk. In the former, each disk stores a small number of redundancy blocks with each set of data blocks, providing a self-sufficient way of recovering from latent sector errors [6]. In the latter, Pergamum computes redundancy information across multiple disks to guard against whole disk failure. However, unlike existing RAID systems, Pergamum can stagger inter-disk activity during data recovery, minimiz-

ing peak energy consumption during rebuilding. Third, energy-efficient decentralized integrity verification is enabled by storing data signatures for disk contents in NVRAM. Thus, using just the signatures, Pergamum tomes can verify the integrity of their local contents and, by exchanging signatures with other Pergamum tomes, verify the integrity of distributed data without incurring any spin up costs. Finally, the Pergamum architecture allows disk-based archives to look like tape: an individual Pergamum tome may be pulled out of the system and read independently; the remaining Pergamum tomes will eventually treat this event like a disk failure and rebuild the “missing” data in a new location.

The goal of Pergamum, is to realize significant cost savings by keeping the vast majority, as many as 95%, of the disks spun down while still providing reasonable performance and excellent reliability. Our techniques allow us to greatly reduce energy usage, as compared to traditional hard drive based systems, making it suitable for archival storage. The use of signatures to verify data reduces the need to power disks on, as does the reduced scrubbing frequency made possible by the extra safety provided by intra-disk parity. Similarly, staggering disk rebuilds reduces peak power load, again allowing Pergamum to reduce the maximum number of disks that must be active at the same time. While we believe these techniques are best realized in a distributed system such as Pergamum—the use of many low-power CPUs is more efficient than a few high-power servers—they are also suitable for use in more conventional MAID architectures, and could be used to reduce power consumption in them as well.

The remainder of this paper is organized as follows. Section 2, places Pergamum within the context of existing research. Following that, Section 3 a detailed discussion of the systems components, including a discussion of the components in each Pergamum tome. Then, Section 4 details the system’s design, including its redundancy and power management approaches. Section 5 contains our evaluation of Pergamum in terms of cost, long-term reliability and performance. Finally, following a discussion of future work in Section 6, we conclude the paper in Section 7.

2 Related Work

In designing Pergamum to meet the goals of energy-efficient, reliable, archival storage [7], we used concepts from various systems. These projects can be distinguished from Pergamum by identifying their intended workload, cost strategy and redundancy strategy. As Table 1 illustrates, many existing systems fulfill some of the goals of Pergamum, but none adequately address all of its concerns.

System	Media	Workload	Redundancy	Consistency	Power Aware
PARAID	disk	server clusters	RAID		Yes
Nomad FS	disk	server clusters	none		Yes
Google File System	disk	data-intensive apps	replicas	relaxed	No
EMC Centera	disk	archival	mirroring or parity	WORM media	No
Venti	disk	archival	RAID 5	content-based naming, type ids	No
Deep Store	disk	archival	selectable replication	content-based naming	No
Copan Revolution 220A	disk	archival	RAID 5	SHA 256	Yes
Sun StorageTek SL8500	tape	backup	N+1	WORM media	No
RAIL	optical	backup, archival	RAID 4	optional write verification	No
Pergamum	disk	archival	2-level erasure coding	algebraic signatures	Yes

Table 1: Overview of storage systems described in Section 2.

A number of systems have also sought to achieve cost savings through the use of commodity hardware [14, 45]. Typically, this strategy assumes that cheaper SATA drives will fail more often than server-class hardware, requiring that the solution utilize additional redundancy techniques. Recent studies, however, call this assumption into question, showing that SATA drives often exhibit the same replacement rate as SCSI and FC disks [37].

Energy efficiency is an area that many designs have explored in pursuit of cost savings. Some reports state that commonly used power supplies operate at only 65–75% efficiency, representing one of the primary culprits of excess heat production, and contributing to cooling demands that account for up to 60% of data-center energy usage [17]. The development of Massive Arrays of Idle Disks (MAIDs) generated large cost savings by leaving the majority of a system’s disks spun down [10]. Further work has expanded on the idea by incorporating strategies such as data migration, the use of drives that can spin at different speeds, and power-aware redundancy techniques [31, 32, 45, 49, 51]. While these systems realize energy savings, they are not designed specifically for archival workloads, instead attempting to provide performance comparable to “full-power” disk arrays at reduced power. Thus, they do not consider approaches that could save even more power at the expense of high performance. For example, some MAID systems, such as those built by Copan Systems [19], use a relatively small number of server-class CPUs and controllers that can control dozens of disks. However, this approach is still relatively power-hungry because the CPU and controllers are always drawing power, reducing energy efficiency. A Copan MAID system in normal use consumes 11 W/TB [19]; as shown in Table 2, this is comparable to the 11–13 W required by a spun-up Pergamum tome with a 1 TB drive. However, it is much higher than the 2–3 W/TB that Pergamum can achieve with 95% of the disks powered off.

Another class of systems relies on media such as tape or optical media rather than hard drives [41, 43]; such

systems are typically used for archival or back-up workloads. While the raw media cost may be somewhat lower than that of disk, the cost savings of such media are often offset by the need for additional hardware, *e. g.*, extra drive heads and robotic arms. Additionally, the random access performance of these systems is often quite poor, introducing a number of correlated side effects such as limitations on the system’s choice of redundancy schemes. For example, RAIL stores data on optical disks and utilizes RAID 4 redundancy, but only at a very high level: for every five DVD libraries, a sixth library is solely devoted to storing parity [43]. Other systems have used striped tape to increase performance [13]; later systems used extra tapes in the stripe to add parity for reliability [24].

A final class of storage systems is designed for an archival workload, but lacks a specific cost-saving strategy [2, 20, 34, 50]. Like many systems designed for primary tier storage, these systems favor performance over power-efficiency and cost savings. While they may offer fast random access performance, their lack of cost efficiency makes them ill-suited for the long-term preservation of large corpora of data. Other wide-area long-term storage systems, such as SafeStore [26], OceanStore [35] and Glacier [21], can provide data longevity, but do not take energy consumption into account. For example, SafeStore uses multiple remote storage systems to ensure data safety, but does not address the issue of reducing power consumption on the remote servers.

Pergamum also expands upon techniques found in systems spanning various usage models and cost strategies. Many systems have used hierarchical hashing as a means of ensuring file integrity [1, 2, 25, 27, 28, 34, 35]; Pergamum extends this technique by utilizing hash trees of algebraic signatures [39]. Additionally, we extend the use of hierarchical hashing to the power-efficient auditing and consistency checking of inter-device replication. Intra-disk redundancy strategies were first suggested for use in full-power RAID systems to avoid loss of data due to disk failure and simultaneous latent sector errors on a surviving disk [11, 12]. Finally, a number of hybrid

drives that combine flash memory with a hard drive have come to market [40]. However, in such units, the flash is used primarily as a read and write cache, in contrast to Pergamum, which uses flash memory on each Pergamum tome for metadata consistency, and indexing information, allowing Pergamum to reduce disk spin-ups while preserving high levels of functionality.

3 System Components

The design of Pergamum was driven by a workload that exhibits read, write and delete behavior that differs from typical disk-based workloads, providing both challenges and opportunities. The workload is write-heavy, motivated by regulatory compliance and the desire to save any data that *might* be valuable at a later date. Reads, while relatively infrequent, are often part of a query or audit and thus are likely to be temporally related. Deletes are also likely to exhibit a temporal relationship as retention policies often specify a maximum data lifetime. This workload resembles traditional archival storage workloads [34, 50], adding deletion for regulatory compliance.

The Pergamum system is structured as a distributed network of independent storage appliances, as shown in Figure 1. Alone, each Pergamum tome acts as an intelligent storage device, utilizing block-level erasure coding to survive media faults and algebraic signatures to verify block integrity. Collectively, the storage appliances provide data reliability through distributed RAID techniques that allow the system to recover from the loss of a device, and inter-disk data integrity by efficiently exchanging hash trees of algebraic signatures. As we will show, this approach is so reliable that disk scrubbing [38] need not be done more frequently than annually. In addition, lost data can be rebuilt with lower peak energy consumption by staggering disk activity; this approach is slower, but reduces peak power consumption.

The next two sections discuss the design and implementation of Pergamum and implementation of these techniques. This section describes an individual Pergamum appliance, or *tome*, including its components, intra-appliance redundancy strategy, interconnection network, and interface. Section 4 then describes how multiple storage appliances work together to provide reliable, distributed, archival storage, including a description of the system’s inter-appliance redundancy and consistency checking strategy.

3.1 Pergamum Tomes

A Pergamum tome is a storage appliance made up of four main components: a low-power processor, a commodity hard drive, non-volatile flash memory and an ethernet

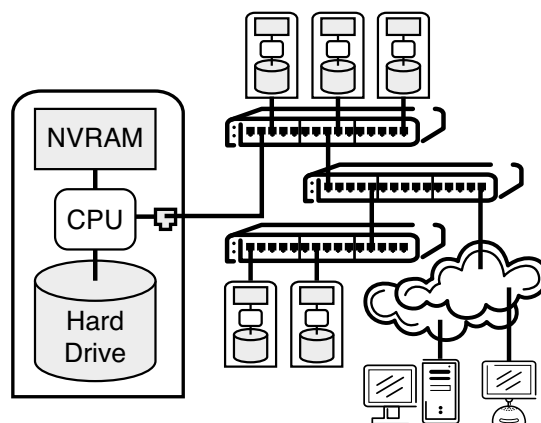


Figure 1: High-level system design of Pergamum. Individual Pergamum tomes, described in Section 3.1 are connected by a commodity network built from off-the-shelf switches.

Component	Power
SATA Hard Drive [47]	7.5 W
ARM-based board (w/ NIC) [4]	3.5 W
NVRAM	< 0.6 W

Table 2: Active power consumption (in watts) of the four primary components that make up a Pergamum tome.

controller. To protect against media errors, erasure coding techniques are used on both the hard drive and flash memory.

Each Pergamum tome is managed by an on-board, low-power CPU; a modern ARM-based single board computer consumes 2–3 W when active (using a 400 MHz CPU) and less than 300 mW when inactive [4]. The processor handles the usual roles required of a network-attached storage device [15, 16] such as network communications, request handling, metadata management, and caching. In addition, each Pergamum tome’s CPU manages consistency checking and parity operations for the local drive, responds to search requests, and initiates communications with other disks to provide inter-disk reliability. The processor can also be used to handle other operations at the device level, such as virus checking and compression.

Persistent storage is provided through the unit’s SATA-class hard drive. The use of commodity hardware offers cost savings over more costly SCSI and FC drives while providing acceptable performance for archival workloads. By using both intra-disk redundancy and distributed redundancy groups, commodity SATA-class drives can provide excellent reliability for long-term archival storage [37].

While a single processor could manage multiple hard drives, Pergamum pairs each processor with a single hard

drive. This is done for performance matching, power savings, and ease of maintenance. As Section 5 details, low-power processors are not fast enough to run even a single disk at full speed, so there is little incentive to control multiple disks with a single CPU. Power savings is another issue: a faster CPU and multi-disk controller would consume more power than multiple individual low-power CPUs (cutting processor voltage in half results in half the clock speed but one fourth the power consumption). Finally, the pairing of a CPU with a single disk and network connection makes it simpler to replace a failed Pergamum tome. If any part of the Pergamum tome fails, the entire Pergamum tome is discarded and replaced, rather than trying to diagnose which part of the Pergamum tome failed to “save” working hard drives. The system then heals itself by rebuilding the data from the failed device elsewhere in the system. By reducing the complexity of routine maintenance, Pergamum reduces ongoing costs.

In addition to a hard drive, each Pergamum tome includes a pool of on-board NVRAM for storing metadata such as the device’s index, data signatures and information about pending writes. The purpose of the NVRAM is to provide low-power, persistent storage; operations such as metadata searches and signature requests do not require the unit’s drive to be spun up. While the use of flash-type NVRAM provides better persistency and energy-efficiency compared to DRAM, it does raise two issues: reliability and durability. Our system protects the flash memory from erroneous writes and media errors through the use of page-level protection and consistency checking [18], ensuring memory reliability. Flash memory is also limited in that the memory must be written in blocks, and each block may only be rewritten a finite number of times, typically 10^4 – 10^5 times. However, since the NVRAM primarily holds metadata such as algebraic signatures and index information, flash writes are relatively rare; flash writes coincide with disk writes. Because this typically occurs fewer than 1000 times per year, or 8000 times during the lifetime of a disk, even if the flash memory is totally overwritten each time, such activity will still be below the 10,000 write cycles that flash memory can support. Additionally, while the current implementation uses NAND flash memory, other technologies such as MRAM [44] and phase change RAM [9] could be used as they become available and price-competitive, further reducing or eliminating the rewrite issue.

Finally, each Pergamum tome includes an Ethernet controller and network port, providing a number of important advantages. First, a network connection is a standardized interface that changes very slowly—modern Ethernet-based systems can interoperate with systems that are more than fifteen years old. In contrast, tape-

based systems require a unique head unit for each tape format, and each of those devices may require a different interface; supporting legacy tapes could require the preservation of lengthy hardware chains. The use of a network also eliminates the need for robotics hardware (or humans) to load and unload media; such robots might need to be modified for different generations of tape media and must be maintained. Instead, the system can use commodity network interconnects, leaving all media permanently connected and always available for messaging.

3.2 Interconnection Network

Since Pergamum must contain thousands of disks to contain the petabytes of data that long-term archives must hold, its network must scale to such sizes. However, throughput is not a major issue for such a network—a modern tape silo with 6,000 tapes typically has fewer than one hundred tape drives, each of which can read or write at about 50 MB/s, for an aggregate throughput of 5 GB/s. Scaling a gigabit Ethernet network to support comparable bandwidth can be done using a star-type network with commodity switches at the “leaves” of the network and, potentially, higher-performance switches in the core. For example, a system built from 48-port gigabit Ethernet switches could use two switches as hubs for 48 switches, each of which supports 46 disks, with the remaining two connections going to each of the two hubs. This approach would support over 2200 disks at minimal cost; if the central hubs each had a few 10 Gb/s uplinks, a single client could easily achieve bandwidth above 5 GB/s. This structure could then be replicated and interconnected using a more expensive 10 Gb/s switch, allowing reasonable-speed access to any one of tens of thousands of drives, with the vast majority remaining asleep to conserve power.

The interconnection network must allow any disk to connect with any network-connected client. By using a standard Ethernet-based network running IP, Pergamum ensures that *any* disk can communicate with any other disk, allowing the system to both detect newly-connected disks and allowing them to communicate with existing disks to “back up” their own data.

The approach described above is highly scalable, with minimal “startup cost” and low incremental cost for adding additional disks. Further efficiencies could be achieved by pairing the Ethernet cable with a higher-gauge wire capable of distributing the 14–18 W that a spun-up disk and processor combination requires. Alternatively, the system could use disks that can spin at variable speeds as low as 5400 RPM [47], reducing disk power requirements to 7.5 W and overall system power needs to below 11 W, sufficiently low to use standard power-over-Ethernet. Central distribution of power has

several advantages, including lower hardware cost and lower cabling cost. Additionally, distributing power via Ethernet greatly simplifies maintenance—adding a new drive simply requires plugging it into an Ethernet cable. While the disks in the system will work to keep average power load below 5% utilization, a central power distribution system will allow the network switches themselves to guarantee that a particular power load will never be exceeded by restricting power distributed by the switch.

3.3 Pergamum Tome Interface

There are two distinct data views in Pergamum: a file-centric view and a block-centric view. Clients utilize the file-centric view, submitting requests to a Pergamum tome through traditional read and write operations. In contrast, requests from one Pergamum tome to another utilize the block-centric view of data based on redundancy group identifiers and offsets.

Clients access data on a Pergamum tome using a set of simple commands and a connection-oriented request and response protocol. Currently, clients address their commands to a specific device, although future versions of Pergamum will include a self-routing communications mechanism. Internally, files are named by a file identifier that is unique within the scope of a single Pergamum tome. The `new` command allocates an unused file identifier and maps it to a filename supplied by the user. This mapping is used by the `open` command to provide the file's unique identifier to a client. This file id, the device's `read` and `write` commands, and a byte offset are then used by the client to access their data.

Requests between Pergamum tomes primarily utilize a data view based on segment identifiers and block offsets, as opposed to files. There are four main operations that take place between Pergamum tomes. First, external parity update requests provide the a Pergamum tome storing parity with the delta and metadata needed to update its external redundancy data. Second, signature requests are used to confirm data integrity. Third, token passing operations assist in determining which devices to spin up. Finally, there are commands for the deferred (*foster*) write operations discussed in Section 4.3.1.

Management of Pergamum tomes can be done either with a centralized “console” to which each Pergamum tome reports its status, or in a distributed fashion where individual Pergamum tomes report their health via LED. For example, each Pergamum tome could have a small green LED that is on when the appliance is working correctly, and off when it is not. An operator would then replace Pergamum tomes whose light is off; this approach is simple and requires little operator skill. Alternatively, a central console could report “Pergamum tome 53 has

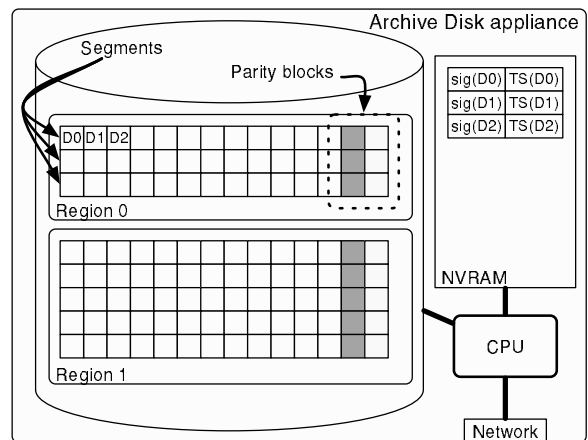


Figure 2: Layout of data on a single Pergamum tome. Data on the disk is divided into blocks and grouped into segments and regions. Data validity is maintained using signatures, and parity blocks are available to rebuild lost or corrupted data.

failed,” triggering a human to replace the failed unit. The Pergamum design permits both approaches; however, we do not discuss the tradeoffs between them in this paper.

4 Pergamum Algorithms and Operation

A Pergamum system, deployed as described in Section 3 is highly decentralized, relying upon individual disks to each manage their own behavior and their own data. Each disk is responsible for ensuring the reliability of the data it stores, using both local redundancy information and storage on other nodes.

4.1 Intra-Disk Storage and Redundancy

The basic unit of storage in a Pergamum tome are fixed-size blocks grouped into fixed-size *segments*, as shown in Figure 2. Together, blocks and segments form the basic units of the system’s two levels of redundancy encoding: intra-disk and inter-disk. Since the system is designed for archival storage, blocks are relatively large—128 KB–1 MB or larger—reducing the metadata overhead necessary to store and index them. This approach mirrors that of tape-based systems, which typically require data to be stored in large blocks to ensure high efficiency and reasonable performance.

The validity of individual blocks is checked using hashes; if a block’s content does not match its hash, it can be identified as incorrect; this approach has been used in other file systems [27, 42]. Disks themselves maintain error-correcting codes, but such codes are insufficiently accurate for long-term archival storage because they have a silent failure rate of about 10^{-14} , a rate sufficiently high

to cause data corruption in large-scale long-term storage. To avoid this problem, each disk appliance stores both a hash value and a timestamp for each block on disk. Assuming a 64-bit hash value and a 32-bit timestamp, a 1 TB disk will require 96 MB of flash memory to maintain this data for 128 KB blocks. Keeping this information in flash memory has several advantages. First, it ensures that block validity information has a different failure mode from the data itself, reducing the likelihood that both data and signature will be corrupted. More importantly, however, it allows the Pergamum tome to access the signatures and timestamps without powering on the disk, enabling Pergamum to conduct inter-disk consistency checks without powering on individual disks.

The hash values used in Pergamum are *algebraic signatures*—hash values that are highly sensitive to small changes in data, but, unlike SHA-1 and RIPEMD, are not cryptographically secure. Algebraic signatures are ideally suited to use in Pergamum because, for many redundancy codes, they exhibit the same relationships that the underlying data does. For example, for simple parity:

$$d_0 \oplus d_1 \cdots \oplus d_{n-1} = p \implies \text{sig}(d_0) \oplus \text{sig}(d_1) \cdots \oplus \text{sig}(d_{n-1}) = \text{sig}(p) \quad (1)$$

While 64-bit algebraic signatures are sufficiently long to reduce the likelihood of “silent” errors to zero; they are ineffective against malicious intruders, though there are approaches to verifying erasure-coded data using signatures or fingerprints that can be used to defeat such attacks [22, 39].

As Figure 2 illustrates, each segment is protected by one or more parity blocks, providing two important protections to improve data survivability. First, the extra parity data provides protection against latent sector errors [6]. If periodic scrubbing reveals unreadable blocks within a segment, the unreadable data can be rebuilt and written to a new block using only the parity on the local disk. Second, while simple scrubbing merely determines whether the block is readable, the use of algebraic signatures and parity blocks allows a disk to determine whether a particular block has been read back properly, catching errors that the disk drive itself cannot [22, 39] and correcting the error without the need to spin up additional disks.

4.2 Inter-Disk Redundancy

While intra-disk parity guards against latent sector errors, Pergamum can survive the loss of an entire Pergamum tome through the use of inter-tome redundancy encoding. Segments on a single disk are grouped into *regions*, and a *redundancy group* is built from regions of identical sizes on multiple disks. To ensure data survival,

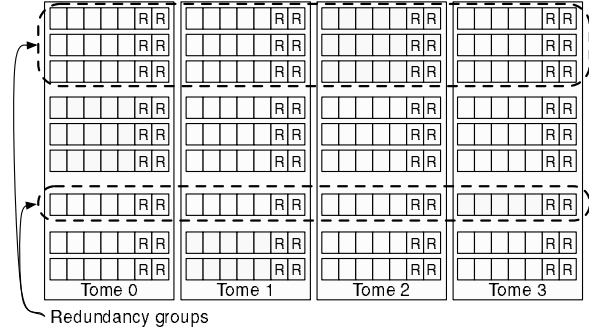


Figure 3: Two levels of redundancy in Pergamum. Individual segments are protected with redundant blocks on the same disk—those labeled with an **R**. Redundancy groups are protected by the shaded segments, which contain erasure correcting codes for the other segments in the redundancy group. Note that segments used for redundancy still contain intra-disk redundant blocks to protect them from latent sector errors.

each redundancy group also includes extra regions on additional disks that contain erasure correction information to allow data to be rebuilt if any disks fail. These *redundancy regions* are stored in the same way as data regions: they have parity blocks to guard against individual block failure and the disk appliances that host them store their algebraic signatures in NVRAM.

The naïve approach to verifying the consistency of a redundancy group would require spinning up all the disks in the group, either simultaneously or in sequence, and verifying that the data in the segments that make up the regions in the group is consistent. Pergamum dramatically reduces this overhead in two ways. First, the algebraic signatures stored in NVRAM can be exchanged between disks in a redundancy group and verified for consistency as described in Section 4.1. Since the signatures are retrieved from NVRAM, the disk need not be spun up during this process as long as changes to on-disk data are reflected in NVRAM. If inconsistencies are found, the timestamps may be used to decide on the appropriate fix. For example, if a set of segments is inconsistent and a data segment is “newer” than the newest parity segment, the problem is likely that the write was not applied properly; depending on how writes have been applied and whether the “old” data is available, the parity may be fixed without powering up the whole set of segments.

While this approach only requires that signatures, rather than data, be transmitted, it is still very inefficient, requiring the transmission of nearly 100 MB of signatures for each disk to verify a redundancy group’s consistency. To further reduce the amount of data and computation that must be done, Pergamum uses hash trees [29] built from algebraic signatures, as shown in Figure 4. Using signatures of blocks as d_i in Equation 1 shows that

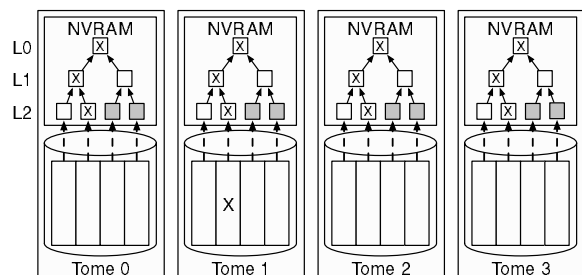


Figure 4: Trees of algebraic signatures. Tomes in a redundancy group exchange the roots of their trees to verify consistency; in this diagram, the signatures marked with an X are inconsistent. The roots (L0) are exchanged; since they do not match, the nodes recurse down the tree to L1 and then L2 to find the source of the inconsistency. “Children” of consistent signatures (signatures shaded in gray at L2) are not fetched, saving transmission and processing time. The inconsistent block on tome 1 is found by checking the intra-segment signatures on each block; only those on tome 1 were inconsistent. Note that only tome 1’s disk need be spun up to identify and correct the error if it is localized.

signatures of sets of signatures follow the same relationships as the underlying data; this property is maintained all the way up to the root of the tree. Thus, the signatures at the roots of each disk’s hash tree for the region should yield a valid erasure code word when combined together. If they do not, some block in the redundancy group is invalid, and the disks recurse down the hash tree to find the bad block, exchanging the contents at each level to narrow the location of the “bad” block. This approach requires $O(k)$ computation and communication when the group is correct—the normal case—and $O(k \log n)$ computation and communication to find an error in a redundancy group with a total of n blocks across k disks. Since redundancy groups are not large ($k \leq 50$, typically), high-level redundancy group verifications can be done quickly and efficiently.

4.3 Disk Power Management

Reducing power consumption is a key goal of Pergamum; since spinning disks are by far the largest consumer of power in a disk appliance, keeping the disk powered off (“spun down”) dramatically reduces power consumption. In contrast to earlier systems that aim to keep 75% of the disks inactive [19], Pergamum tries to keep 95% or more of the disks inactive all of the time, reducing disk power consumption by a factor of five or more over existing MAID approaches. This goal is achieved with several strategies: sequentially activating disks to update redundancy information on writes, low-frequency scrubbing, and sequentially rebuilding regions

on failed disks.

To guard against too many disks being spun up at once, Pergamum uses *spin-up tokens*, which are passed from one node to another to allow spin-up. If multiple nodes require a token simultaneously, the node currently holding the token (which may or may not be spun up at the time) calculate need based on factors such as a unit’s oldest pending request, the types of requests it has pending, the number of pending requests and the last time the disk was spun up.

4.3.1 Reading and Writing Data

When a client requests a data read, the device from which data is to be read is spun up. This process takes a few seconds, after which data can be read at full speed. While a Pergamum tome is somewhat slower than a high-power network-attached disk, its performance, discussed in Section 5, is sufficient for archival storage retrieval. Moreover, since the data is stored on a disk rather than a tape, random access performance is significantly better than that of a tape-based system.

As with reads, archive writes require a spun-up disk. Pergamum clients choose the disks to which they write data; Pergamum does not impose a choice on users. This is done because some clients may want to group particular data on specific disks: for example, a company might choose to archive email for an individual user on one drive. On the other hand, a storage client may query Pergamum nodes to identify spun-up nodes, allowing it to select a disk that is already spun up.

Since writes require the eventual update of distributed data, they are more involved than reads. First, the target disk is spun up if it is not already active. Next, data is written to blocks on the local disk. However, existing data blocks are not overwritten in place; instead, data is written to a new data block, allowing the Pergamum tome to calculate “deltas” based on the old and new block. These deltas are then sent to the Pergamum tomes storing the redundancy regions for the old block’s segment. On the local device, the segment mapping is updated to replace the old block with the new block. It is important to note, however, that the old block is retained until it has been confirmed that all external parity has been updated.

On the Pergamum tomes storing the redundancy information, the deltas arrive as a parity update request. Since the redundancy update destination knows how the erasure correcting code is calculated, it can use the delta from the data target disk to update its own redundancy information; it does not need both the old and new data block, only the delta. Because the delta may be different for different parity disks, however, the Pergamum tome that received the original write request must keep both old and new data until all of the parity segments have

been updated. However, doing updates this way ensures that a write requires no more than two disks to be active at any time; while the total energy to write the data is unchanged—a write to an (m, n) redundancy group must still update $n - m + 1$ disks—the peak energy is dramatically reduced from $n - m + 1$ disks active to 2 disks active, resulting in an improvement for any code that can correct more than one erasure.

One problem with allowing writes directed to a specific Pergamum tome is that the disk may not be spun up when the write is issued. While the destination disk may be activated, an alternate approach is to write the data to *any* currently active disk and later copy the data to the “correct” destination. This approach is called *surrogate writing*, and is used in Pergamum to avoid spinning disks up too frequently. Instead, writes are directed to an already-active disk, and the Pergamum tome to which data will eventually be sent is also notified. The data can then be transferred to the correct destination lazily.

4.3.2 Scrubbing and Recovering Data

To ensure reliability, disks in Pergamum are occasionally scrubbed: every block on the disk is read and checked for agreement with the signature stored in NVRAM. This procedure is relatively time-consuming; even at 10 MB/s, a 1 TB disk requires more than a day to check. However, Pergamum tome’s use of on-disk redundancy to guard the data in a segment, described in Section 4.1, greatly reduces the danger of data loss from latent sector errors, so the system can reduce the frequency with which it performs full-disk scrubs. Instead, a Pergamum tome performs a “limited scrub” each time it is spun up, either during idle periods or immediately before the disk is spun down. This limited scrub checks a few hundred randomly-chosen locations on the disk for correctness and examines the drive’s SMART status [5], ensuring that the disk is basically operating correctly. If the drive passes this check, the major concern is total drive failure, either during operation or during spin-up, as Section 5.2 describes.

Complete drive failures are handled by rebuilding the data on the lost drive in a new location. However, since fewer than 5% of the disks in Pergamum may be on at any given time and redundancy groups that may contain data and parity on 15–40 disks for maximal storage efficiency, it is impractical to spin up all of the disks in a redundancy group to rebuild it. Instead, Pergamum uses techniques similar to those used in writing data to recover data lost when a disk fails. The rebuilding algorithm begins by choosing a new location for the data that has been lost; this may be on an existing disk (as long as it is not already part of the redundancy region), or it may be on a newly-added disk. Pergamum then spins up the disks in

the redundancy region one by one, with each disk sending its data to the node on which data is being rebuilt. The node doing the rebuilding folds the incoming data into the data already written using the redundancy algorithm; thus, it must write each location in the region m times and read it $m - 1$ times (the first “read” would result in all zeros, and is skipped).

5 Experimental Evaluation

Our experiments with the current implementation of Pergamum were designed to measure several things. First, we wanted to evaluate the cost of our system in order to ensure that our solution was economically feasible. Second, we wanted to confirm that Pergamum can provide long-term reliability through a strategy of multiple levels of parity and consistency checking using algebraic signatures. Finally, we wanted to measure the performance of our implementation to show that Pergamum is suitable for archival workloads and to identify potential bottlenecks.

The remainder of this section proceeds as follows. First, we first present an analytical evaluation of the system’s cost. Then, using a series of simulations, we examine the system’s long-term reliability. Finally, we present the results of our performance tests with the current implementation of Pergamum.

5.1 Cost

An archival system’s cost can be broken down into two primary areas: static (initial costs) and operational. The first figure describes the cost to acquire the system, and the second figure quantifies the cost to run the system. Examining both costs together is important because low static costs can be overshadowed by the total cost of operating and maintaining a system over its lifetime.

We do not consider personnel costs in any of the systems we describe; we assume that all of the systems are sufficiently well automated that human maintenance costs are relatively low. However, this assumption is somewhat optimistic, especially for large tape-based systems that use complex hardware that may require repair. In contrast, Pergamum is built from simple, disposable components—a failed Pergamum tome or network switch may simply be thrown out rather than repaired, reducing the time and personnel effort required to maintain the system.

Static costs reflect the expenses associated with acquiring an archival storage solution, and can be calculated by totaling a number of individual costs. One is the system expense, which totals the base hardware and software costs of a storage system with a given capacity for storage media. This cost is paid at least once per

storage system, regardless of how much storage is actually required. Media cost, in dollars per terabyte, is a second expense. Large archival storage systems may require several “base” systems; for example, an archival system that uses tape silos and robots might require one silo per 6,000 tape cartridges, even if the silo will not be filled initially.

Operational costs reflect those costs incurred by day to day operation of an archival storage system. This cost can be measured using a dollars per operational period figure, normalized to the amount of storage being managed. Some of the primary contributors to a system’s total operational expenses include power, cooling, floor space and management. As described above, we omit management cost, both because we assume it will be similar for different storage technologies, and because it is extremely difficult to quantify. We also omit the cost of floor space since it is highly variable depending on the location of the data center. However, an important, but often omitted, aspect of operational costs includes the expenses related to reliability: expected replacement costs for failed media and the operational cost associated with parity operations. This cost, along with power and cooling, forms the basis of our comparison of operational costs.

The static and operational costs must include the cost for any redundant hardware or storage. However, since existing solutions vary in their reliability, even within a particular technology, we have not attempted to quantify the interplay between capacity and reliability. Instead, we assume that a system that requires mirroring simply costs twice as much to purchase and run per byte as a non-redundant system. In this respect, Pergamum is very low cost: the storage overhead for a system with segments using 62 data and 2 parity blocks and redundancy groups with 13 data disks and 3 parity disks is $\frac{64}{62} \times \frac{16}{13} - 1 = 0.27$ times usable data capacity. In such a system, 1 TB of raw storage can hold 787 GB of user data.

All of these factors—static cost, operational cost, and redundancy overhead—are summarized in Table 3. Static costs are approximations based on publically available hardware prices. For operational costs, We have used a constant rate of \$0.20/kWh for electricity to cover both the direct cost of power and the cost of cooling. Table 3 shows the costs for a 10 PB archive for each technology, including sufficient base systems to reach this capacity. While the costs reflected in the table are approximate, they are useful for comparative purposes. Also, we note that some systems have ranges for redundancy overhead because they can be configured in several ways to ensure sufficient reliability; we chose the least expensive reliability option for each technology. For example, the EMC Centera [20] can be used with mirroring; doing so

might increase reliability, but will certainly increase total cost.

The results summarized in Table 3 illustrate a number of cost-related archival storage issues. First, as shown by PARAID, even energy-efficient, non-archival systems are too expensive for archival scenarios. Second, media with low storage densities can become expensive very quickly because they require a large amount of hardware to manage the high numbers of media. For example, RAIL uses UDO2 optical media that only offers 60 GB per disk and thus the system requires numerous cabinets and drives to handle the volume of media. Using off-the-shelf dual-layer DVDs, with capacity under 10 GB per disk, would reduce the media cost, but would increase the hardware cost by a factor of six because of the added media; such an approach would require 100 DVDs per terabyte, making the cost prohibitive. Third, the Copan and Centera demonstrate two different strategies for cost effective storage: lower initial costs versus lower runtime costs. Finally, we see that Pergamum is competitive in cost to Sun’s StorageTek SL8500 system while providing functionality, such as inter-archive redundancy, that tape-based systems are unable to provide.

An understanding of the costs associated with reliability is important because it assists in matching the data to be protected with an economically efficient reliability strategy. Unfortunately, because it is largely dependent on the data itself, the economic impact of lost data is difficult to calculate. Moreover, many of the costs resulting from data loss are, at best, difficult to quantify. For example, the cost to replace data can vary from zero (don’t replace it) to nearly priceless (how much is bank account data worth?). Another factor, opportunity costs, expresses the cost of lost time; every hour spent dealing with data loss is an hour that is not spent doing something else. In a professional setting, data loss may also involve mandatory disclosures that could introduce costs associated with bad publicity and fines. While we do not quantify these costs, we note that long-term archive reliability is a serious issue [7].

5.2 Reliability

There are many tradeoffs that influence the reliability of an archival storage system. Factors such as stripe size, both on an individual disk and between disks, disk failure rate, disk rebuild time and the expected rate of latent sector errors must be considered when building a long-term archival system. Our analysis considers these factors along with strict power-management constraints to compute the expected mean time to data loss (MTTDL) of a deployed Pergamum system.

Table 4 shows the parameters used in our analysis. In the absence of an archival workload for our reliability

System	Media	Static cost	Oper. cost	Redundancy
Sun StorageTek SL8500	T10000 tape	\$4,250	\$60	None
EMC Centera	SATA HD	\$6,600	\$1,800	parity
PARAID	SCSI HD	\$37,800	\$1,200	RAID
Copan Revolution	SATA HD	\$19,000	\$250	RAID-5
RAIL	UDO2	\$57,000	\$225	RAID-4 (5+1)
Pergamum	SATA HD	\$4,700	\$50	2-level

Table 3: Comparison of system and operational costs for 10 PB of storage. All costs are in thousands of dollars and reflect common configurations. Operational costs were calculated assuming energy costs of \$0.20/kWh (including cooling costs).

Parameter	Value
Disk Fail Rate (λ_D)	1/100000 hours
Disk Repair Rate (μ_D)	1/100 hours
Latent Sector Fault Rate (λ_S)	1/13245 hours
Scrub Rate (μ_S)	1/8640 hours

Table 4: Simulator and model parameters.

analysis, we assume that each active device transfers a constant 2 MB/s, on average. Given a byte error rate of 1×10^{-14} , the on-disk sector error rate is approximately 1/13245 hours. Due to the incremental nature of our rebuild algorithm, we approximate the time to rebuild a single device in our system to be roughly 100 hours, or 3 MB/s. Finally each disk in the system fails at a rate of 1/100000 hours and is subject to a full scrub every year or 8640 hours. We consider these estimates to be liberal and provide a near-worst-case MTTDL of our system.

In order to determine the reliability of our system, we developed a discrete event simulator in Python using the SimPy module. There are four core events in our simulator: DiskFail, DiskRebuild, SectorFail and Scrub. Values for disk failure time, sector failure time and disk scrub are all drawn from an exponential distribution, while disk rebuild takes place in simulation time at 3 MB/s. We model the effects of disk spin-up by subtracting 10 hours from the life of a disk every time it is spun up [38]; this may well be pessimistic, resulting in an MTTDL that is shorter than in a real system. Each iteration of the simulator runs until a data loss event is reached and the current time is recorded. Although we found that around 100 iterations is sufficient, we calculate the MTTDL of a single configuration by running 1000 iterations with that configuration.

We also use Markov models to compute the reliability of single, double and triple disk fault tolerant codes. These models only capture disk failure and rebuild, and thus serve two purposes. First, the models give us a straightforward way to verify the behavior of the simulator. Most importantly, the MTTDL computed from each model serves as an approximation to a system that has

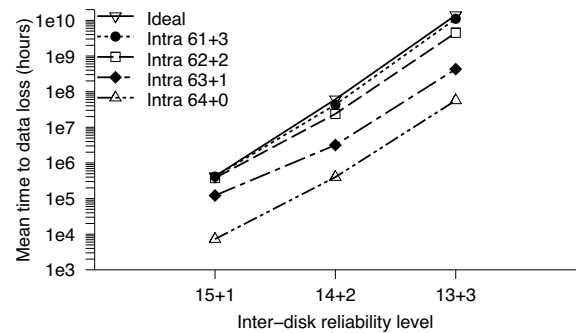


Figure 5: Mean time to data loss in hours for a single 16 disk group. 61+3 intra-disk parity is nearly equivalent to the “ideal” system, in which latent sector errors never occur. Note that MTTDL of 10^{10} hours for 16 disks corresponds to a 1000 year MTTDL for a 10 PB Pergamum system.

the ability to handle any number of latent sector errors.

Recent work has shown that latent sector errors make a non-trivial contribution to system reliability [6]. We thus modeled data loss in our system for configurations with 1, 2, and 3 parity segments per redundancy group under several different assumptions: levels of intra-disk parity protection ranging from 0–3 parity blocks per segment, and an “ideal” analytical model which assumed *no* latent sector errors occurred and considered only whole-disk failures. The results of our modeling using a scrub rate of once per year for each disk, shown in Figure 5, indicate that latent sector errors do indeed cause data loss if nothing is done to guard against them. The distance between the top curve (“ideal” MTTDL without latent sector errors) and bottom curve (no intra-disk parity) is approximately two orders of magnitude, showing that Pergamum must guard against data loss from latent sector errors. However, by using intra-disk erasure coding, the effect of latent sectors on MTTDL is nearly eliminated. In essence, we are trading disk space for a longer scrub interval, saving power in the process. Figure 5 shows that a configuration of 3 intra-disk parity blocks per 64 block segment provides nearly two or-

ders of magnitude longer MTTDL than no protection at all, approaching the “ideal” situation where latent sector errors never exist. With the exception of the configurations with 3 inter-disk parity elements and the “ideal” case, all of the MTTDL values in the graph are based on 1000 iterations of the simulator; we were only able to capture tens of MTTDL numbers for the configurations involving 3 inter-disk parity elements and 1 or 2 intra-parity elements, and the simulation for 3 inter-disk parity and 3 intra-disk parity elements took a great deal of time to run and only resulted in a few data points. This lack of data is due to the extremely high reliability of these configurations—the simulator modeled many failures, but so few caused data loss that the simulation ran very slowly. This behavior is precisely what we want from an archival storage system: it can gracefully handle many failure events without losing data. Even though we captured fewer data points for the triple inter-parity configuration, we believe the reported MTTDL is a reasonable approximation. As we see in the graph, the use of 3 intra-disk parity elements is close to the “ideal” situation across all inter-parity configurations.

Our simulation and modeling show that a configuration of 3 inter-disk parity segments per 16-disk reliability group and 3 intra-disk parity blocks per segment will result in an MTTDL of approximately 10^{10} hours. If each disk has a raw capacity of 1 TB, a Pergamum system capable of storing 10 PB of user data will require about 800 such groups, resulting an MTTDL of 1.25×10^7 hours, or about 1,400 years. Should this MTTDL for an entire archive be too low, we would recommend using more inter-disk parity—3 parity blocks per 64 block segment can correct most of the latent sector errors.

5.3 Performance

The current Pergamum prototype system consists of approximately 1,400 lines of Python 2.5 code, with an additional 300 lines of C code that were used to implement performance-sensitive operations such as data encoding and low-level disk operations. Our implementation includes the core system functionality, including internal redundancy, external redundancy, and a client interface that allows for basic I/O interactions. In its current state however, the implementation relies upon statically assigned redundancy groups and it does not include scrubbing or consistency checking.

For testing, all systems were located on the same gigabit Ethernet switch with little outside contention for computing or network resources. Communication between the Pergamum tome and the client used standard TCP/IP sockets in Python. For maximum compatibility, we utilized an MTU size of 1500 B.

Each Pergamum tome was equipped with an ARM 9

Test	Client	Server
Raw Data Transfer	20.02	20.96
Raw Data Write	9.33	9.98
Unsafe Pergamum Write	4.74	4.74
XOR Parity Pergamum Write	4.72	3.25
Reed Solomon Pergamum Write	4.25	1.67
Fully Protected Pergamum Write	3.66	0.75
Pergamum Read	5.77	5.78

Table 5: Read and write performance for a single Pergamum tome to client connection. XOR parity writes used 63 data blocks to one parity block segments. Reed Solomon writes used 62 data blocks to two parity block segments. Fully protected writes utilize two level of Reed Solomon encoding and the server throughput reflects time to fully encode and commit internal and external parity updates.

CPU running at 400 MHz, 128 MB of DDR2 SDRAM and Linux version 2.6.12.6. The client was equipped with an Intel Core Duo processor running at 2 GHz, 2 GB of DDR2 SDRAM and OS X version 10.4.10. The primary storage on each Pergamum tome was provided by a 7200 RPM SATA drive formatted with XFS. For read and write performance experiments, we utilized block sizes of 1 MB and 64 blocks per segment. Persistent metadata storage utilized a 1 GB USB flash drive and Berkeley DB version 4.4. The workload consisted of randomly generated files, all several megabytes in size.

5.3.1 Read and Write Throughput

Our first experiment with the Pergamum implementation was an evaluation of the device’s raw data transfer performance. As Table 5 shows, the maximum throughput of a single TCP/IP stream to a Pergamum tome is 20 MB/s at the device. Further tests showed that, the device could copy data from a network buffer to an on-disk file at about 10 MB/s. Together, these values serve as an upper limit for the write performance that could be expected from a single client connection over TCP/IP.

Write throughput using the Pergamum software layer was tested at varying levels of write safety. The first write test was conducted with no internal or external parity updates. As shown in Table 5, writes without data protection ran at 4.74 MB/s. While no redundancy encoding was performed in the unsafe write, the system did incur the overhead of updating segment metadata and dividing the incoming data into fixed-size blocks.

Testing with internal parity updates enabled was performed using both simple XOR-based parity and more advanced Reed Solomon encoding. In these tests, the client-side and server-side throughput differ, as Pergamum utilizes parity logging during writes. Thus, while

the client views throughput as the time taken to simply ingest the data, the Pergamum tome's throughput includes the time to ingest the data and update the redundancy information. The first test utilized simple XOR-based parity in a 63+1 (63 data blocks and 1 parity block) configuration. This arrangement achieved a client-side write throughput of 4.72 MB/s and a Pergamum tome-side throughput of 3.25 MB/s. As Table 5 shows, using Reed Solomon in a 62+2 configuration results in similar client side throughput, 4.25 MB/s. However, the extra processing and parity block updates results in a server throughput of 1.67 MB/s.

The final write test, fully protected Pergamum tome writes, utilizes both inter- and intra-disk redundancy. Internal parity utilized Reed Solomon encoding in a 62+2 configuration. External redundancy utilized Reed Solomon with 3 data regions to 2 parity regions. In this configuration, client throughput is reduced to 3.66 MB/s as the CPU is taxed with both internal and external parity calculations. This is evident in the server throughput which is reduced to 0.75 MB/s. However, this does reflect the time required to update both internal and external parity and thus reflects the rate at which a single Pergamum tome can protect data with full internal and external parity.

Profile data obtained from the test runs indicates the system is CPU-bound. The performance penalty for the Pergamum tome writes appears to be based largely on two factors. First, as shown in the difference between a raw write and an unsafe Pergamum tome write in Table 5, Python's buffer management imposes a performance penalty, an issue that could be remedied with an optimized, native implementation. Second, as seen in the difference between the XOR Pergamum tome write and the Reed Solomon write, data encoding imposes a significant penalty for lower power processors. This is further evident by the results of our read throughput tests. Since a read operation to the Pergamum tome involves less buffer management and parity operations, throughput is correspondingly faster. We were able to achieve sustained read rates of 5.78 MB/s.

While the performance numbers in Table 5 would be inadequate for most high-performance workloads, even our current, prototype implementation of Pergamum is capable of supporting archival workloads. For example, 1000 Pergamum tomes and a spin-up rate of only 5% can provide a system-level ingestion throughput in excess of 175 MB/s, ingesting a terabyte in 90 minutes and fully protecting it in 8 hours. At this rate such an archive built from 1 TB disks could be filled in a year.

Encode Operations	ARM9	Core Duo
XOR parity	20.02	201.41
Reed Solomon; 5 data, 2 parity	3.13	33.68
Data signature (64-bit)	57.44	533.33

Table 6: Throughput, in MB/sec, to encode 50 MB of data using the Pergamum tome's 400 MHz ARM9 board drawing 2-3 W and a desktop class 2 GHz Intel Core Duo drawing 31 W.

5.3.2 Data Encoding

One of the primary functions of each Pergamum tome's processor is data encoding for redundancy and signature generation. Thus, we wanted to confirm that the low-power CPUs used by Pergamum to save energy are actually capable of meeting the encoding demands of archival workloads.

In our first data encoding test, we measured the throughput of the XOR operation by updating parity for 50 MB of data. We were able to achieve an average encoding rate of 20.79 MB/s on the tome's CPU. For reference, a desktop class processor using the same library was able to encode data at 201.41 MB/s. However, this performance increase comes at the cost of power consumption; the Intel Core Duo processor consumes 31 W compared to the tome's ARM-based processor which consumes roughly 2.5 W for the entire board.

A similar result was achieved when updating parity for 50 MB of data protected by a 5+2 Reed Solomon configuration. As Table 6 summarizes, the processor on the Pergamum tome was able to encode the new parity blocks at a rate of 3.13 MB/s. For reference, the desktop processor could encode at average rate of 33.68 MB/s. Again, we notice an order of magnitude throughput increase at the cost of over an order of magnitude power consumption increase.

Our final encoding experiment involved the generation of data signatures. Our current implementation of Pergamum generates data signatures using $GF(2^{32})$ arithmetic in an optimized C-based library. Generating 64 bit signatures over 32 bit symbols, we achieved an average signature generation throughput of 57.44 MB/s. For reference, the same library on the desktop-class client achieved a rate of 533.33 MB/s.

Our results indicate that the low-power processor on the Pergamum tome is capable of encoding data at a rate comparable to its power consumption. Additionally, we believe that is capable of adequately encoding data for an archival system's write-once, read maybe usage model. While our current performance numbers are reasonable, our experience in designing and implementing the Pergamum prototype has shown that low-power processors greatly benefit from carefully optimized code.

Our early implementations provided more than adequate performance on a desktop class computer but were somewhat slow on the Pergamum tome's low-power CPU.

6 Future Work

While Pergamum demonstrates some of the features needed in an archival storage system, work remains to turn it into a fully effective, evolving, long-term storage system. In addition to the engineering tasks associated with optimizing the Pergamum implementation for low-power CPUs, there are a number of important research areas to examine.

Storage management in Pergamum, and archival storage in general, is an open area with a number of interesting problems. Management strategies play a large part in cost efficiency; many believe that management costs eclipse hardware costs [3]. As a long-term data repository, the effectiveness of archival storage is increased as management overhead is decreased and, ideally, automated; the easier it is to store long-term data, the more ubiquitous it will become. Thus, Pergamum must address how extensibility can be handled in an automatic way, without sacrificing its distributed nature, or the independence of its Pergamum tomes. This overall question includes a number of facets. How are redundancy groups populated? How does the system know if a Pergamum tome is nonfunctional as opposed to temporarily off-line? How can the system's capacity be expanded while still providing adequate reliability?

In our current implementation, users interact with Pergamum by submitting requests to specific Pergamum tomes using a connection-oriented protocol. In future versions, the use of a simple, standardized `put` and `get` style protocol, such as that provided by HTTP, could allow storage to be more evolvable and permit the use of standard tools for storing and retrieving information. Further, techniques such as distributed searching that take into account data movement and migration could greatly simplify how users interact with the system.

While the trade-off between redundancy and storage usage is well acknowledged, there is still work to be done in understanding the interplay of redundancy, storage overhead and power consumption. We have chosen a relatively small set of points in this space; future work could explore this space more completely. This could include an examination of which redundancy codes are best suited to the unique demands and usage model of archival storage.

Long-term storage systems must assume that no single device will serve as the storage appliance for the data's entire lifetime. Thus, data migration in a secure and power-efficient manner is another requirement for Pergamum, and is a critical area for research. This re-

search direction also has implications for reliability; a policy of device refreshment could be an integral part of a long-term reliability strategy.

The optimality of the choice of one CPU and network connection per disk is also an open question; our choice is based on both quantitative and qualitative factors, but other arrangements are certainly possible. Additionally, it has always been assumed that client machines would include modern desktop level CPUs that could be leveraged for pre-processing. Similarly, determining the best network to use to connect thousands of (mostly idle) devices is an interesting problem to consider.

7 Conclusions

We have developed Pergamum, a system designed to provide reliable, cost-effective archival storage using low-power, network-attached disk appliances. Reliability is provided through two levels of redundancy encoding: intra-disk redundancy allows an individual device to automatically rebuild data in the event of small-scale data corruption, while inter-disk redundancy provides protection from the loss of an entire device. Fixed costs are kept low through the use of a standardized network interface, and commodity hardware such as SATA drives; since each Pergamum tome is essentially "disposable", a system operator can simply throw away faulty nodes. Operational costs are controlled by utilizing ultra-low-power CPUs, power-managed disks and new techniques such as local NVRAM for caching metadata and redundancy information to avoid disk spin-ups, intra-disk redundancy, staggered data rebuilding, and hash trees of algebraic signatures for distributed consistency checking. Finally, Pergamum's performance is acceptable for archival storage: the use of many low-power CPUs instead of a few server-class CPUs results in disks that can transfer data at 3–5 MB/s, with faster performance possible through the use of optimized code.

At 2–3 W/TB and under \$0.50/GB for a full system, Pergamum is far cheaper and more reliable than existing MAID systems, though the techniques we have developed may be applied to more conventional MAID designs as well. Moreover, a Pergamum system is comparable in cost and energy consumption to a large-scale tape archive, while providing much higher reliability, faster random access performance and better manageability. The combination of low power usage, low hardware cost, very high reliability, simpler management, and excellent long-term upgradability make Pergamum a strong choice for storage in long-term data archives.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback on the ideas in this paper, helping us to refine them. We would also like to thank our shepherd Doug Terry and the anonymous reviewers for their insightful comments that helped us improve the paper.

This research was supported by the Petascale Data Storage Institute under Department of Energy award DE-FC02-06ER25768, and by the industrial sponsors of the SSRC, including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Agami Systems, Data Domain, Digisense, Hewlett-Packard Laboratories, IBM Research, LSI Logic, Network Appliance, Seagate, Symantec, and Yahoo!.

References

- [1] ADYA, A., BOLOSKEY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), USENIX.
- [2] AGARWALA, S., PAULY, A., RAMACHANDRAN, U., AND SCHWAN, K. e-SAFE: An extensible, secure and fault tolerant storage system. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)* (2007), pp. 257–268.
- [3] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: running circles around storage administration. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002).
- [4] ARCOM, INC. <http://www.arcom.com/>, Aug. 2007.
- [5] ATA SMART feature set commands. Small Form Factors Committee SFF-8035. <http://www.t13.org>.
- [6] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (June 2007).
- [7] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006* (Apr. 2006), pp. 221–234.
- [8] CAO, P., LIN, S. B., VENKATARAMAN, S., AND WILKES, J. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems* 12, 3 (1994), 236–269.
- [9] CHEN, Y. C., RETTNER, C. T., RAOUX, S., BURR, G. W., CHEN, S. H., SHELBY, R. M., SALINGA, M., RISK, W. P., HAPP, T. D., MCCLELLAND, G. M., BREITWISCH, M., SCHROTT, A., PHILIPP, J. B., LEE, M. H., CHEEK, R., NIRSCHL, T., LAMOREY, M., CHEN, C. F., JOSEPH, E., ZAIDI, S., YEE, B., LUNG, H. L., BERGMANN, R., AND LAM, C. Ultra-thin phase-change bridge memory device using GeSb. In *International Electron Devices Meeting (IEDM '06)* (Dec. 2006), pp. 1–4.
- [10] COLARELLI, D., AND GRUNWALD, D. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)* (Nov. 2002).
- [11] DHOLAKIA, A., ELEFThERIOU, E., HU, X.-Y., ILIADIS, I., MENON, J., AND RAO, K. Analysis of a new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. In *Proceedings of the 2006 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2006), pp. 373–374.
- [12] DHOLAKIA, A., ELEFThERIOU, E., HU, X.-Y., ILIADIS, I., MENON, J., AND RAO, K. Analysis of a new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. Tech. Rep. RZ 3652, IBM Research, Mar. 2006.
- [13] DRAPEAU, A. L., AND KATZ, R. H. Striped tape arrays. Tech. Rep. CSD-93-730, Computer Science Division, University of California, Berkeley, 1993.
- [14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, NY, Oct. 2003), ACM.
- [15] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, Oct. 1998), pp. 92–103.
- [16] GIBSON, G. A., AND VAN METER, R. Network attached storage architecture. *Communications of the ACM* 43, 11 (2000), 37–45.
- [17] GREEN GRID CONSORTIUM. The green grid opportunity, decreasing datacenter and other IT energy usage patterns. <http://www.thegreengrid.org>, Feb 2007.
- [18] GREENAN, K. M., AND MILLER, E. L. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *6th ACM & IEEE Conference on Embedded Software (EMSOFT '06)* (Seoul, Korea, Oct. 2006), ACM.
- [19] GUHA, A. Solving the energy crisis in the data center using CO-PAN Systems' enhanced MAID storage platform. Copan Systems white paper, Dec. 2006.
- [20] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing commodity storage clusters. In *Proceedings of the 32nd Int'l Symposium on Computer Architecture* (June 2005), pp. 60–71.
- [21] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005), USENIX.
- [22] HENDRICKS, J., GANGER, G. R., AND REITER, M. K. Verifying distributed erasure-coded data. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC 2007)* (Aug. 2007).
- [23] Health Information Portability and Accountability Act, Oct. 1996.
- [24] HUGHES, J., MILLIGAN, C., AND DEBIEZ, J. High performance RAIT. In *Proceedings of the 19th IEEE Symposium on Mass Storage Systems and Technologies* (Apr. 2002), pp. 65–73.
- [25] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Mar. 2003), USENIX, pp. 29–42.

- [26] KOTLA, R., ALVISI, L., AND DAHLIN, M. SafeStore: a durable and practical storage system. In *Proceedings of the 2007 USENIX Annual Technical Conference* (June 2007), pp. 129–142.
- [27] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004).
- [28] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems* 23, 1 (2005), 2–50.
- [29] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology - Crypto '87* (Berlin, 1987), Springer-Verlag, pp. 369–378.
- [30] OXLEY, M. G. (H.R.3763) Sarbanes-Oxley Act of 2002, Feb. 2002.
- [31] PINHEIRO, E., AND BIANCHINI, R. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th International Conference on Supercomputing* (June 2004).
- [32] PINHEIRO, E., BIANCHINI, R., AND DUBNICKI, C. Exploiting redundancy to conserve energy in storage systems. In *Proceedings of the 2006 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Saint Malo, France, June 2006).
- [33] PRESTON, W. C., AND DIDIO, G. Disk at the price of tape? an in-depth examination. Copan Systems white paper, 2004.
- [34] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, California, USA, 2002), USENIX, pp. 89–101.
- [35] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (Mar. 2003), pp. 1–14.
- [36] SAITO, Y., FRÖLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004), pp. 48–58.
- [37] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2007), pp. 1–16.
- [38] SCHWARZ, T. J. E., XIN, Q., MILLER, E. L., LONG, D. D. E., HOSPODOR, A., AND NG, S. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)* (Oct. 2004), IEEE, pp. 409–418.
- [39] SCHWARZ, S. J., T., AND MILLER, E. L. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)* (Lisboa, Portugal, July 2006), IEEE.
- [40] SEAGATE TECHNOLOGY LLC. Momentus 5400 psd. http://www.seagate.com/docs/pdf/marketing/ds_momentus_5400_psd.pdf, Aug 2007.
- [41] SUN MICROSYSTEMS. Sun StorageTek SL8500 modular library system. http://www.sun.com/storagetek/tape_storage/tape_libraries/sl8500/.
- [42] SUN MICROSYSTEMS. Solaris ZFS and Red Hat Enterprise Linux EXT3 file system performance. http://www.sun.com/software/whitepapers/solaris10/zfs_linux.pdf, June 2007.
- [43] TANABE, T., TAKAYANAGI, M., TATEMITI, H., URA, T., AND YAMAMOTO, M. Redundant optical storage system using DVD-RAM library. In *Proceedings of the 16th IEEE Symposium on Mass Storage Systems and Technologies* (Mar. 1999), pp. 80–87.
- [44] TEHRANI, S., SLAUGHTER, J. M., CHEN, E., DURLAM, M., SHI, J., AND DEHERRERA, M. Progress and outlook for MRAM technology. *IEEE Transactions on Magnetics* 35, 5 (Sept. 1999), 2814–2819.
- [45] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A.-I. A., REIHER, P., AND KUENNING, G. PARAD: A gear-shifting power-aware RAID. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2007).
- [46] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, Nov. 2006), USENIX.
- [47] WESTERN DIGITAL. WD Caviar GP 1 TB SATA hard drives. <http://www.westerndigital.com/en/library/sata/2879-701229.pdf>, Aug. 2007.
- [48] WILCKE, W. W., GARNER, R. B., FLEINER, C., FREITAS, R. F., GOLDING, R. A., GLIDER, J. S., KENCHAMMANA-HOSEKOTE, D. R., HAFNER, J. L., MOHIUDDIN, K. M., RAO, K., BECKER-SZENDY, R. A., WONG, T. M., ZAKI, O. A., HERNANDEZ, M., FERNANDEZ, K. R., HUELS, H., LENK, H., SMOLIN, K., RIES, M., GOETTERT, C., PICUNKO, T., RUBIN, B. J., KAHN, H., AND LOO, T. IBM Intelligent Bricks project—petabytes and beyond. *IBM Journal of Research and Development* 50, 2/3 (2006), 181–197.
- [49] YAO, X., AND WANG, J. RIMAC: a novel redundancy-based hierarchical cache architecture for energy efficient, high performance storage systems. In *Proceedings of EuroSys 2006* (Oct. 2006), pp. 249–262.
- [50] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, Apr. 2005), IEEE.
- [51] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WILKES, J. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)* (Brighton, UK, Oct. 2005), ACM.

Scalable Performance of the Panasas Parallel File System

Brent Welch¹, Marc Unangst¹, Zainul Abbasi¹, Garth Gibson^{1,2}, Brian Mueller¹,
Jason Small¹, Jim Zelenka¹, Bin Zhou¹

¹Panasas, Inc. ²Carnegie Mellon

{welch,mju,zabbasi,garth,bmueller,jsmall,jimz,bzhou}@panasas.com

Abstract

The Panasas file system uses parallel and redundant access to object storage devices (OSDs), per-file RAID, distributed metadata management, consistent client caching, file locking services, and internal cluster management to provide a scalable, fault tolerant, high performance distributed file system. The clustered design of the storage system and the use of client-driven RAID provide scalable performance to many concurrent file system clients through parallel access to file data that is striped across OSD storage nodes. RAID recovery is performed in parallel by the cluster of metadata managers, and declustered data placement yields scalable RAID rebuild rates as the storage system grows larger. This paper presents performance measures of I/O, metadata, and recovery operations for storage clusters that range in size from 10 to 120 storage nodes, 1 to 12 metadata nodes, and with file system client counts ranging from 1 to 100 compute nodes. Production installations are as large as 500 storage nodes, 50 metadata managers, and 5000 clients.

1 Introduction

Storage systems for high performance computing environments must be designed to scale in performance so that they can be configured to match the required load. Clustering techniques are often used to provide scalability. In a storage cluster, many nodes each control some storage, and the overall distributed file system assembles the cluster elements into one large, seamless storage system. The storage cluster can be hosted on the same computers that perform data processing, or they can be a separate cluster that is devoted entirely to storage and accessible to the compute cluster via a network protocol.

The Panasas storage system is a specialized storage cluster, and this paper presents its design and a number of performance measurements to illustrate the scalability. The Panasas system is a production system that provides file service to some of the largest compute clusters in the world, in scientific labs, in seismic data processing, in digital animation studios, in computational fluid dynamics, in semiconductor

manufacturing, and in general purpose computing environments. In these environments, hundreds or thousands of file system clients share data and generate very high aggregate I/O load on the file system. The Panasas system is designed to support several thousand clients and storage capacities in excess of a petabyte.

The unique aspects of the Panasas system are its use of per-file, client-driven RAID, its parallel RAID rebuild, its treatment of different classes of metadata (block, file, system) and a commodity parts based blade hardware with integrated UPS. Of course, the system has many other features (such as object storage, fault tolerance, caching and cache consistency, and a simplified management model) that are not unique, but are necessary for a scalable system implementation.

2 Panasas File System Background

This section makes a brief tour through the system to provide an overview for the following sections. The two overall themes to the system are object storage, which affects how the file system manages its data, and clustering of components, which allows the system to scale in performance and capacity.

The storage cluster is divided into storage nodes and manager nodes at a ratio of about 10 storage nodes to 1 manager node, although that ratio is variable. The storage nodes implement an object store, and are accessed directly from Panasas file system clients during I/O operations. The manager nodes manage the overall storage cluster, implement the distributed file system semantics, handle recovery of storage node failures, and provide an exported view of the Panasas file system via NFS and CIFS. Figure 1 gives a basic view of the system components.

2.1 Object Storage

An *object* is a container for data and attributes; it is analogous to the inode inside a traditional UNIX file system implementation. Specialized storage nodes called Object Storage Devices (OSD) store objects in a local OSDFS file system. The object interface addresses objects in a two-level (partition ID/object ID)

namespace. The OSD wire protocol provides byte-oriented access to the data, attribute manipulation, creation and deletion of objects, and several other specialized operations [OSD04]. We use an iSCSI transport to carry OSD commands that are very similar to the OSDv2 standard currently in progress within SNIA and ANSI-T10 [SNIA].

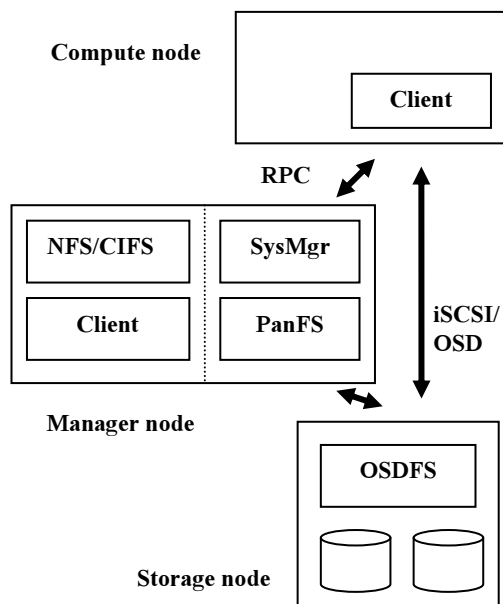


Figure 1: Panasas System Components

The Panasas file system is layered over the object storage. Each file is striped over two or more objects to provide redundancy and high bandwidth access. The file system semantics are implemented by metadata managers that mediate access to objects from clients of the file system. The clients access the object storage using the iSCSI/OSD protocol for Read and Write operations. The I/O operations proceed directly and in parallel to the storage nodes, bypassing the metadata managers. The clients interact with the out-of-band metadata managers via RPC to obtain access capabilities and location information for the objects that store files. The performance of striped file access is presented later in the paper.

Object attributes are used to store file-level attributes, and directories are implemented with objects that store name to object ID mappings. Thus the file system metadata is kept in the object store itself, rather than being kept in a separate database or some other form of storage on the metadata nodes. Metadata operations are described and measured later in this paper.

2.2 System Software Components

The major software subsystems are the OSDFS object storage system, the Panasas file system metadata manager, the Panasas file system client, the NFS/CIFS gateway, and the overall cluster management system.

- The Panasas client is an installable kernel module that runs inside the Linux kernel. The kernel module implements the standard VFS interface, so that the client hosts can mount the file system and use a POSIX interface to the storage system. We don't require any patches to run inside the 2.4 or 2.6 Linux kernel, and have tested with over 200 Linux variants.
- Each storage cluster node runs a common platform that is based on FreeBSD, with additional services to provide hardware monitoring, configuration management, and overall control.
- The storage nodes use a specialized local file system (OSDFS) that implements the object storage primitives. They implement an iSCSI target and the OSD command set. The OSDFS object store and iSCSI target/OSD command processor are kernel modules. OSDFS is concerned with traditional block-level file system issues such as efficient disk arm utilization, media management (i.e., error handling), high throughput, as well as the OSD interface.
- The cluster manager (SysMgr) maintains the global configuration, and it controls the other services and nodes in the storage cluster. There is an associated management application that provides both a command line interface (CLI) and an HTML interface (GUI). These are all user level applications that run on a subset of the manager nodes. The cluster manager is concerned with membership in the storage cluster, fault detection, configuration management, and overall control for operations like software upgrade and system restart [Welch07].
- The Panasas metadata manager (PanFS) implements the file system semantics and manages data striping across the object storage devices. This is a user level application that runs on every manager node. The metadata manager is concerned with distributed file system issues such as secure multi-user access, maintaining consistent file- and object-level metadata, client cache coherency, and recovery from client, storage node, and metadata server crashes. Fault tolerance is based on a local transaction log that is replicated to a backup on a different manager node.
- The NFS and CIFS services provide access to the file system for hosts that cannot use our Linux installable file system client. The NFS service is a tuned version of the standard FreeBSD NFS server that runs inside the kernel. The CIFS service is based on Samba and runs at user level. In turn, these services use a local instance of

the file system client, which runs inside the FreeBSD kernel. These *gateway* services run on every manager node to provide a clustered NFS and CIFS service.

2.3 Commodity Hardware Platform

The storage cluster nodes are implemented as *blades* that are very compact computer systems made from commodity parts. The blades are clustered together to provide a scalable platform. Up to 11 blades fit into a 4U (7 inches) high shelf chassis that provides dual power supplies, a high capacity battery, and one or two 16-port GE switches. The switches aggregate the GE ports from the blades into a 4 GE trunk. The 2nd switch provides redundancy and is connected to a 2nd GE port on each blade. The battery serves as a UPS and powers the shelf for a brief period of time (about five minutes) to provide orderly system shutdown in the event of a power failure. Any number of blades can be combined to create very large storage systems.

The OSD *StorageBlade* module and metadata manager *DirectorBlade* module use the same form factor blade and fit into the same chassis slots. The StorageBlade module contains a commodity processor, two disks, ECC memory, and dual GE NICs. The DirectorBlade module has a faster processor, more memory, dual GE NICs, and a small private disk. In addition to metadata management, DirectorBlades also provide NFS and CIFS service, and their large memory is used as a data cache when serving these protocols. Details of the different blades used in the performance experiments are given in Appendix I.

Any number of shelf chassis can be grouped into the same storage cluster. A shelf typically has one or two DirectorBlade modules and 9 or 10 StorageBlade modules. A shelf with 10 StorageBlade modules contains 5 to 15 TB of raw storage in 4U of rack space. Customer installations range in size from 1 shelf to around 50 shelves, although there is no enforced limit on system size.

While the hardware is essentially a commodity PC (i.e., no ASICs), there are two aspects of the hardware that simplified our software design. The first is the integrated UPS in the shelf chassis that makes all of main memory NVRAM. The metadata managers do fast logging to memory and reflect that to a backup with low latency network protocols. OSDFS buffers write data so it can efficiently manage block allocation. The UPS powers the system for several minutes to protect the system as it shuts down cleanly after a power failure. The metadata managers flush their logs to a local disk, and OSDFS flushes writes through to disk.

The logging mechanism is described and measured in detail later in the paper. The system monitors the battery charge level, and will not allow a shelf chassis to enter service without an adequately charged battery to avoid data loss during back-to-back power failures.

The other important aspect of the hardware is that blades are a Field Replaceable Unit (FRU). Instead of trying to repair a blade, if anything goes wrong with the hardware, the whole blade is replaced. We settled on a two-drive storage blade as a compromise between cost, performance, and reliability. Having the blade as a failure domain simplifies our fault tolerance mechanisms, and it provides a simple maintenance model for system administrators. Reliability and data reconstruction are described and measured in detail later in the paper.

3 Storage Management

Traditional storage management tasks involve partitioning available storage space into LUNs (i.e., logical units that are one or more disks, or a subset of a RAID array), assigning LUN ownership to different hosts, configuring RAID parameters, creating file systems or databases on LUNs, and connecting clients to the correct server for their storage. This can be a labor-intensive scenario. We sought to provide a simplified model for storage management that would shield the storage administrator from these kinds of details and allow a single, part-time admin to manage systems that were hundreds of terabytes in size.

The Panasas storage system presents itself as a file system with a POSIX interface, and hides most of the complexities of storage management. Clients have a single mount point for the entire system. The `/etc/fstab` file references the cluster manager, and from that the client learns the location of the metadata service instances. The administrator can add storage while the system is online, and new resources are automatically discovered. To manage available storage, we introduced two basic storage concepts: a physical storage pool called a *BladeSet*, and a logical quota tree called a *Volume*.

The BladeSet is a collection of StorageBlade modules in one or more shelves that comprise a RAID fault domain. We mitigate the risk of large fault domains with the scalable rebuild performance described in Section 4.2. The BladeSet is a hard physical boundary for the volumes it contains. A BladeSet can be grown at any time, either by adding more StorageBlade modules, or by merging two existing BladeSets together.

The Volume is a directory hierarchy that has a quota constraint and is assigned to a particular BladeSet. The quota can be changed at any time, and capacity is not allocated to the Volume until it is used, so multiple volumes compete for space within their BladeSet and grow on demand. The files in those volumes are distributed among all the StorageBlade modules in the BladeSet.

Volumes appear in the file system name space as directories. Clients have a single mount point for the whole storage system, and volumes are simply directories below the mount point. There is no need to update client mounts when the admin creates, deletes, or renames volumes.

Each Volume is managed by a single metadata manager. Dividing metadata management responsibility along volume boundaries (i.e., directory trees) was done primarily to keep the implementation simple. We figured that administrators would introduce volumes (i.e., quota trees) for their own reasons, and this would provide an easy, natural boundary. We were able to delay solving the multi-manager coordination problems created when a parent directory is controlled by a different metadata manager than a file being created, deleted, or renamed within it. We also had a reasonable availability model for metadata manager crashes; well-defined subtrees would go offline rather than a random sampling of files. The file system recovery check implementation is also simplified; each volume is checked independently (and in parallel when possible), and errors in one volume don't affect availability of other volumes. Finally, clients bypass the metadata manager during read and write operations, so the metadata manager's load is already an order of magnitude smaller than a traditional file server storing the same number of files. This reduces the importance of fine-grain metadata load balancing. That said, uneven volume utilization can result in uneven metadata manager utilization. Our protocol allows the metadata manager to redirect the client to another manager to distribute load, and we plan to exploit this feature in the future to provide finer-grained load balancing.

While it is possible to have a very large system with one BladeSet and one Volume, and we have customers that take this approach, we felt it was important for administrators to be able to configure multiple storage pools and manage quota within them. Our initial model only had a single storage pool: a file would be partitioned into component objects, and those objects would be distributed uniformly over all available storage nodes. Similarly, metadata management would

be distributed by randomly assigning ownership of new files to available metadata managers. This is similar to the Ceph model [Weil06]. The attraction of this model is smooth load balancing among available resources. There would be just one big file system, and capacity and metadata load would automatically balance. Administrators wouldn't need to worry about running out of space, and applications would get great performance from large storage systems.

There are two problems with a single storage pool: the fault and availability model, and performance isolation between different users. If there are ever enough faults to disable access to some files, then the result would be that a random sample of files throughout the storage system would be unavailable. Even if the faults were transient, such as a node or service crash and restart, there will be periods of unavailability. Instead of having the entire storage system in one big fault domain, we wanted the administrator to have the option of dividing a large system into multiple fault domains, and of having a well defined availability model in the face of faults. In addition, with large installations the administrator can assign different projects or user groups to different storage pools. This isolates the performance and capacity utilization among different groups.

Our storage management design reflects a compromise between the performance and capacity management benefits of a large storage pool, the backup and restore requirements of the administrator, and the complexity of the implementation. In practice, our customers use BladeSets that range in size from a single shelf to more than 20 shelves, with the largest production Bladeset being about 50 shelves, or 500 StorageBlade modules and 50 DirectorBlade modules. The most common sizes, however, range from 5 to 10 shelves. While we encourage customers to introduce Volumes so the system can better exploit the DirectorBlade modules, we have customers that run large systems (e.g., 20 shelves) with a single Volume.

3.1 Automatic Capacity Balancing

Capacity imbalance occurs when expanding a BladeSet (i.e., adding new, empty storage nodes), merging two BladeSets, and replacing a storage node following a failure. In the latter scenario, the imbalance is the result of our RAID rebuild, which uses spare capacity on every storage node rather than dedicating a specific "hot spare" node. This provides better throughput during rebuild (see section 4.2), but causes the system to have a new, empty storage node after the failed storage node is replaced. Our system automatically

balances used capacity across storage nodes in a BladeSet using two mechanisms: *passive balancing* and *active balancing*.

Passive balancing changes the probability that a storage node will be used for a new component of a file, based on its available capacity. This takes effect when files are created, and when their stripe size is increased to include more storage nodes. Active balancing is done by moving an existing component object from one storage node to another, and updating the storage map for the affected file. During the transfer, the file is transparently marked read-only by the storage management layer, and the capacity balancer skips files that are being actively written. Capacity balancing is thus transparent to file system clients.

Capacity balancing can serve to balance I/O load across the storage pool. We have validated this in large production systems. Of course there can always be transient hot spots based on workload. It is important to avoid long term hot spots, and we did learn from some mistakes. The approach we take is to use a uniform random placement algorithm for initial data placement, and then preserve that during capacity balancing. The system must strive for a uniform distribution of both objects and capacity. This is more subtle than it may appear, and we learned that biases in data migration and placement can cause hot spots.

Initial data placement is uniform random, with the components of a file landing on a subset of available storage nodes. Each new file gets a new, randomized storage map. However, the uniform random distribution is altered by passive balancing that biases the creation of new data onto emptier blades. On the surface, this seems reasonable. Unfortunately, if a single node in a large system has a large bias as the result of being replaced recently, then it can end up with a piece of every file created over a span of hours or a few days. In some workloads, recently created files may be hotter than files created several weeks or months ago. Our initial implementation allowed large biases, and we occasionally found this led to a long-term hot spot on a particular storage node. Our current system bounds the effect of passive balancing to be within a few percent of uniform random, which helps the system fine tune capacity when all nodes are nearly full, but does not cause a large bias that can lead to a hot spot.

Another bias we had was favoring large objects for active balancing because it is more efficient. There is per-file overhead to update its storage map, so it is more efficient to move a single 1 GB component object

than to move 1000 1 MB component objects. However, consider a system that has relatively few large files that are widely striped, and lots of other small files. When it is expanded from N to $N+M$ storage nodes (e.g., grows from 50 to 60), should the system balance capacity by moving a few large objects, or by moving many small objects? If the large files are hot, it is a mistake to bias toward them because the new storage nodes can get a disproportionate number of hot objects. We found that selecting a uniform random sample of objects from the source blades was the best way to avoid bias and inadvertent hot spots, even if it means moving lots of small objects to balance capacity.

4 Object RAID and Reconstruction

We protect against loss of a data object or an entire storage node by striping files across objects stored on different storage nodes, using a fault-tolerant striping algorithm such as RAID-1 or RAID-5. Small files are mirrored on two objects, and larger files are striped more widely to provide higher bandwidth and less capacity overhead from parity information. The per-file RAID layout means that parity information for different files is not mixed together, and easily allows different files to use different RAID schemes alongside each other. This property and the security mechanisms of the OSD protocol [Gobioff97] let us enforce access control over files even as clients access storage nodes directly. It also enables what is perhaps the most novel aspect of our system, client-driven RAID. That is, the clients are responsible for computing and writing parity. The OSD security mechanism also allows multiple metadata managers to manage objects on the same storage device without heavyweight coordination or interference from each other.

Client-driven, per-file RAID has four advantages for large-scale storage systems. First, by having clients compute parity for their own data, the XOR power of the system scales up as the number of clients increases. We measured XOR processing during streaming write bandwidth loads at 7% of the client's CPU, with the rest going to the OSD/iSCSI/TCP/IP stack and other file system overhead. Moving XOR computation out of the storage system into the client requires some additional work to handle failures. Clients are responsible for generating good data and good parity for it. Because the RAID equation is per-file, an errant client can only damage its own data. However, if a client fails during a write, the metadata manager will scrub parity to ensure the parity equation is correct.

The second advantage of client-driven RAID is that clients can perform an end-to-end data integrity check.

Data has to go through the disk subsystem, through the network interface on the storage nodes, through the network and routers, through the NIC on the client, and all of these transits can introduce errors with a very low probability. Clients can choose to read parity as well as data, and verify parity as part of a read operation. If errors are detected, the operation is retried. If the error is persistent, an alert is raised and the read operation fails. We have used this facility to track down flakey hardware components; we have found errors introduced by bad NICs, bad drive caches, and bad customer switch infrastructure. While file systems like ZFS [ZFS] maintain block checksums within a local file system, which does not address errors introduced during the transit of information to a network client. By checking parity across storage nodes within the client, the system can ensure end-to-end data integrity. This is another novel property of per-file, client-driven RAID.

Third, per-file RAID protection lets the metadata managers rebuild files in parallel. Although parallel rebuild is theoretically possible in block-based RAID, it is rarely implemented. This is due to the fact that the disks are owned by a single RAID controller, even in dual-ported configurations. Large storage systems have multiple RAID controllers that are not interconnected. Since the SCSI Block command set does not provide fine-grained synchronization operations, it is difficult for multiple RAID controllers to coordinate a complicated operation such as an online rebuild without external communication. Even if they could, without connectivity to the disks in the affected parity group, other RAID controllers would be unable to assist. Even in a high-availability configuration, each disk is typically only attached to two different RAID controllers, which limits the potential speedup to 2x.

When a StorageBlade module fails, the metadata managers that own Volumes within that BladeSet determine what files are affected, and then they farm out file reconstruction work to every other metadata manager in the system. Metadata managers rebuild their own files first, but if they finish early or do not own any Volumes in the affected Bladeset, they are free to aid other metadata managers. Declustered parity groups [Holland92] spread out the I/O workload among all StorageBlade modules in the BladeSet. The result is that larger storage clusters reconstruct lost data more quickly. Scalable reconstruction performance is presented later in this paper.

The fourth advantage of per-file RAID is that unrecoverable faults can be constrained to individual files. The most commonly encountered double-failure

scenario with RAID-5 is an unrecoverable read error (i.e., grown media defect) during the reconstruction of a failed storage device. The 2nd storage device is still healthy, but it has been unable to read a sector, which prevents rebuild of the sector lost from the first drive and potentially the entire stripe or LUN, depending on the design of the RAID controller. With block-based RAID, it is difficult or impossible to directly map any lost sectors back to higher-level file system data structures, so a full file system check and media scan will be required to locate and repair the damage. A more typical response is to fail the rebuild entirely. RAID controllers monitor drives in an effort to scrub out media defects and avoid this bad scenario, and the Panasas system does media scrubbing, too. However, with high capacity SATA drives, the chance of encountering a media defect on drive B while rebuilding drive A is still significant. With per-file RAID-5, this sort of double failure means that only a single file is lost, and the specific file can be easily identified and reported to the administrator. While block-based RAID systems have been compelled to introduce RAID-6 (i.e., fault tolerant schemes that handle two failures), we have been able to deploy highly reliable RAID-5 systems with large, high performance storage pools.

4.1 RAID I/O Performance

This section shows I/O performance as a function of the size of the storage system, the number of clients, and the striping configuration. Streaming I/O and random I/O performance are shown.

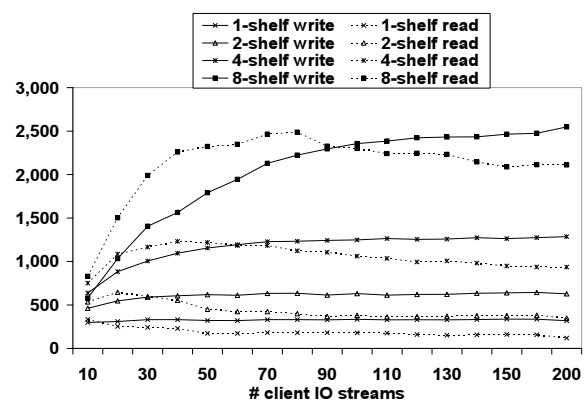


Figure 2: IOzone Streaming Bandwidth MB/sec

Figure 2 charts iozone [Iozone] streaming bandwidth performance from a cluster of up to 100 clients against storage clusters of 1, 2, 4 and 8 shelves. Each client ran two instances of iozone writing and reading a 4GB file with 64KB record size. (Note that the X-axis is not

linear; there is a jump from 160 I/O streams to 200.) Appendix I summarizes the details of the hardware used in the experiments.

This is a complicated figure, but there are two basic results. The first is that performance increases linearly as the size of the storage system increases. The second is that write performance scales up and stays flat as the number of clients increases, while the read performance tails off as the number of clients increases. The write performance curves demonstrate the performance scalability. A one-shelf system delivered about 330 MB/sec, a two-shelf system delivered about 640 MB/sec, a four-shelf system delivered about 1280 MB/sec, and the eight-shelf system peaked around 2500 MB/sec. This corresponds to a scaling factor that is 95% of linear. In another experiment, a 30-shelf system achieved just over 10 GB/sec of read performance, for a per-shelf bandwidth of 330 MB/sec.

These kinds of results depend on adequate network bandwidth between clients and the storage nodes. They also require a 2-level RAID striping pattern for large files to avoid network congestion [Nagle04]. For a large file, the system allocates parity groups of 8 to 11 storage nodes until all available storage nodes have been used. Approximately 1 GB of data (2000 stripes) is stored in each parity group before rotating to the next one. When all parity groups have been used, the file wraps around to the first group again. The system automatically selects the size of the parity group so that an integral number of them fit onto the available storage nodes with the smallest unused remainder. The 2-level RAID pattern concentrates I/O on a small number of storage nodes, yet still lets large files expand to cover the complete set of storage nodes. Each file has its own mapping of parity groups to storage nodes, which diffuses load and reduces hot-spotting.

The difference between read and write scaling stems from the way OSDFS writes data. It performs delayed block allocation for new data so it can be batched and written efficiently. Thus new data and its associated metadata (i.e., indirect blocks) are streamed out to the next available free space, which results in highly efficient utilization of the disk arm. Read operations, in contrast, must seek to get their data because the data sets are created to be too large to fit in any cache. While OSDFS does object-aware read ahead, as the number of concurrent read streams increases, it becomes more difficult to optimize the workload because the amount of read-ahead buffering available for each stream shrinks.

Figure 3 charts iiozone performance for mixed (i.e., read and write) random I/O operations against a 4 GB file with different transfer sizes and different numbers of clients. Each client has its own file, so the working set size increases with more clients. Two parameters were varied: the amount of memory on the StorageBlade modules, and the I/O transfer size. The vertical axis shows the throughput of the storage system, and the chart compares the different configurations as the number of clients increases from 1 to 6. The results show that larger caches on the StorageBlade modules can significantly improve the performance of small block random I/O.

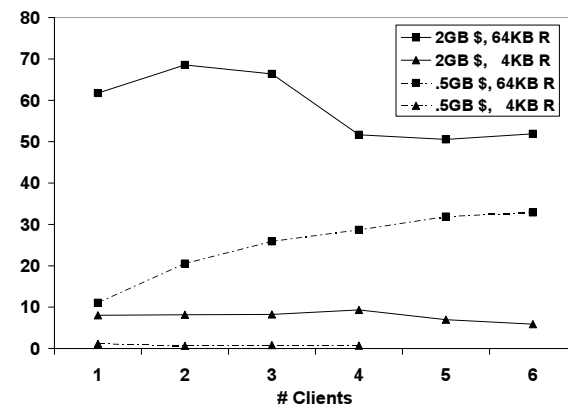


Figure 3: Mixed Random I/O MB/sec

We tested two different hardware configurations: StorageBlade modules with 512 MB of memory (labeled as “.5GB \$”) and with 2 GB of memory (labeled “2GB \$”). In each case the system had 9 StorageBlade modules, so the total memory on the StorageBlade modules was 4.5 GB and 18 GB, respectively. Two different transfer sizes are used: 64 KB matches the stripe unit size, and 4 KB is the underlying block size of OSDFS. Obviously, the larger memory configuration is able to cache most or all of the working set with small numbers of clients. As the number of clients increases such that the working set size greatly exceeds the cache, then the difference in cache size will matter less. The throughput with 4 KB random I/O is very low with inadequate cache. One client gets approximately 1.1 MB/sec, or about 280 4 KB ops/sec, and the rate with 4 clients drops to 700 KB/sec, or about 175 ops/sec. The 4 KB and 64 KB writes in the mixed workload require four OSD operations to complete the RAID-5 update to the full stripe (two reads, two writes). In addition, we observed extra I/O traffic between the client cache and the OSD due to read ahead and write gathering optimizations that are enabled by default to optimize streaming workloads. The iiozone test does 1 million I/Os from each client in the 4 KB block and 4 GB file case, so we elected not to

run that with 5 and 6 clients in the 512 MB cache configuration simply because it ran too long.

4.2 RAID Rebuild Performance

RAID rebuild performance determines how quickly the system can recover data when a storage node is lost. Short rebuild times reduce the window in which a second failure can cause data loss. There are three techniques to reduce rebuild times: reducing the size of the RAID parity group, *declustering* the placement of parity group elements, and rebuilding files in parallel using multiple RAID engines.

The rebuild bandwidth is the rate at which reconstructed data is written to the system when a storage node is being reconstructed. The system must read N times as much as it writes, depending on the width of the RAID parity group, so the overall throughput of the storage system is several times higher than the rebuild rate. A narrower RAID parity group requires fewer read and XOR operations to rebuild, so will result in a higher rebuild bandwidth. However, it also results in higher capacity overhead for parity data, and can limit bandwidth during normal I/O. Thus, selection of the RAID parity group size is a tradeoff between capacity overhead, on-line performance, and rebuild performance.

Understanding declustering is easier with a picture. In Figure 4, each parity group has 4 elements, which are indicated by letters placed in each storage device. They are distributed among 8 storage devices. The ratio between the parity group size and the available storage devices is the *declustering ratio*, which in this example is $\frac{1}{2}$. In the picture, capital letters represent those parity groups that all share the 2nd storage node. If the 2nd storage device were to fail, the system would have to read the surviving members of its parity groups to rebuild the lost elements. You can see that the other elements of those parity groups occupy about $\frac{1}{2}$ of each other storage device.

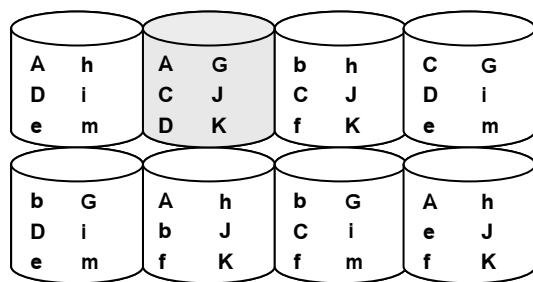


Figure 4: Declustered parity groups

For this simple example you can assume each parity element is the same size so all the devices are filled equally. In a real system, the component objects will have various sizes depending on the overall file size, although each member of a parity group will be very close in size. There will be thousands or millions of objects on each device, and the Panasas system uses active balancing to move component objects between storage nodes to level capacity.

Declustering means that rebuild requires reading a subset of each device, with the proportion being approximately the same as the declustering ratio. The total amount of data read is the same with and without declustering, but with declustering it is spread out over more devices. When writing the reconstructed elements, two elements of the same parity group cannot be located on the same storage node. Declustering leaves many storage devices available for the reconstructed parity element, and randomizing the placement of each file's parity group lets the system spread out the write I/O over all the storage. Thus declustering RAID parity groups has the important property of taking a fixed amount of rebuild I/O and spreading it out over more storage devices.

Having per-file RAID allows the Panasas system to divide the work among the available DirectorBlade modules by assigning different files to different DirectorBlade modules. This division is dynamic with a simple master/worker model in which metadata services make themselves available as workers, and each metadata service acts as the master for the volumes it implements. By doing rebuilds in parallel on all DirectorBlade modules, the system can apply more XOR throughput and utilize the additional I/O bandwidth obtained with declustering.

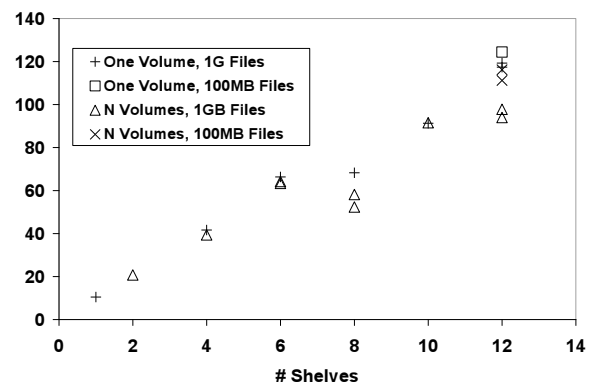


Figure 5: RAID Rebuild MB/sec vs. System Size

Figure 5 plots rebuild performance as the size of the storage cluster grows from 1 DirectorBlade module and

10 StorageBlade modules up to 12 DirectorBlade modules and 120 StorageBlade modules. Each shelf has 1 DirectorBlade module (1.6 GHz Xeon) and 10 StorageBlade modules. In this experiment, the system was populated with 100 MB files or 1 GB files, and each glyph in the chart represents an individual test. The declustering ratio ranges from 0.9 to 0.075, and the resulting reconstruction bandwidth ranges from 10 MB/sec to 120 MB/sec. Declustering and parallel rebuild gives nearly linear increase in rebuild performance as the system gets larger.

The reduced performance at 8 and 10 shelves stems from a wider stripe size. The system automatically picks a stripe width from 8 to 11, maximizing the number of storage nodes used while leaving at least one spare location. For example, in a single-shelf system with 10 StorageBlade modules and 1 distributed spare, the system will use a stripe width of 9. The distributed spare allows reconstruction to proceed without replacing a failed storage node; each file's storage map skips at least one available storage node, creating a virtual spare location for that file that can be used to store a rebuilt copy of a failed component. Each file has its own spare location, which distributes the spares across the Bladeset. The system reserves capacity on each storage node to allow reconstruction. With 80 storage nodes and 1 distributed spare, the system chooses a stripe width of 11 so that 7 parity groups would fit, leaving 3 unused storage nodes. A width of 10 cannot be used because there would be no unused storage nodes. Table 1 lists the size of the parity group (i.e., stripe width) as a function of the size of the storage pool.

Storage Nodes	Group Width	Groups
10	9	1
20	9	2
40	9	4
60	8	7
80	11	7
100	11	9
120	9	13

Table 1: Default Parity Group Size

The variability in the 12-shelf result came from runs that used 1 GB files and multiple Volumes. In this test, the number of files impacted by the storage node failure varied substantially among Volumes because only 30 GB of space was used on each storage node, and each metadata manger only had to rebuild between 25 and 40 files. There is a small delay between the time a metadata manager completes its own Volume and the

time it starts working for other metadata managers; as a result, not every metadata manager is fully utilized towards the end of the rebuild. When rebuilding a nearly full StorageBlade, this delay is insignificant, but in our tests it was large enough to affect the results. Since we compute bandwidth by measuring the total rebuild time and dividing by the amount of data rebuilt, this uneven utilization skewed the results lower. We obtained higher throughput with less variability by filling the system with 10 times as many 100 MB files, which results in a more even distribution of files among Volume owners, or by using just a single Volume to avoid the scheduling issue.

Figure 6 shows the effect of RAID parity group width on the rebuild rate. If a parity stripe is 6-wide, then the 5 surviving elements are read to recompute the missing 6th element. If a parity stripe is only 3-wide, then only 2 surviving elements are read to recompute the missing element. Even though the reads can be issued in parallel, there is more memory bandwidth associated with reads, and more XOR work to do with the wider stripe. Therefore narrower parity stripes are rebuilt more quickly. The experiment confirms this.

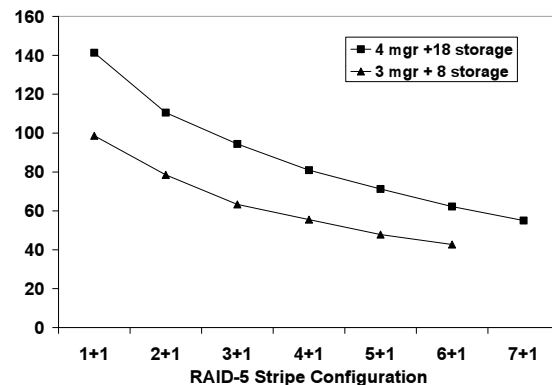


Figure 6: RAID Rebuild MB/sec vs Stripe Width

We measured two systems. One had three DB-100 DirectorBlade modules, and 8 SB-500a-XC StorageBlade modules. The maximum stripe width in this configuration is 7 to allow for the spare. The other system had four DB-100a DirectorBlade modules and 18 SB-500a-XC StorageBlade modules in two shelves. The maximum stripe width in this configuration was 8. Rebuild bandwidth increases with narrower stripes because the system has to read less data to reconstruct the same amount. The results also show that having more DirectorBlade modules increases rebuild rate. This is because there are more “reconstruction engines” that can better exploit the bandwidth available in the system. These results indicate that the rebuild performance of the large systems shown in Figure 5

could be much higher with 2 DirectorBlade modules per shelf, more than twice the performance shown since those results used older, first generation DirectorBlade modules.

5 Metadata Management

There are several kinds of metadata in our system. These include the mapping from object IDs to sets of block addresses, mapping files to sets of objects, file system attributes such as ACLs and owners, file system namespace information (i.e., directories), and configuration/management information about the storage cluster itself. One approach might be to pick a common mechanism, perhaps a relational database, and store all of this information using that facility. This shifts the issues of scalability, reliability, and performance from the storage system over to the database system. However, this makes it more difficult to optimize the metadata store for the unique requirements of each type of metadata. In contrast, we have provided specific implementations for each kind of metadata. Our approach distributes the metadata management among the object storage devices and metadata managers to provide scalable metadata management performance, and allows selecting the best mechanism for each metadata type.

5.1 Block-level Metadata

Block-level metadata is managed internally by OSDFS, our file system that is optimized to store objects. OSDFS uses a floating block allocation scheme where data, block pointers, and object descriptors are batched into large write operations. The write buffer is protected by the integrated UPS, and it is flushed to disk on power failure or system panics. Our block allocation algorithms are similar to those of WAFL [Hitz94] and LFS [Rosenblum90], although unlike LFS there is no cleaner that compacts free space. Fragmentation was an issue in early versions of OSDFS that used a first-fit block allocator, but this has been significantly mitigated in later versions that use a modified best-fit allocator.

OSDFS stores higher level file system data structures, such as the partition and object tables, in a modified BTree data structure. Block mapping for each object uses a traditional direct/indirect/double-indirect scheme. Free blocks are tracked by a proprietary bitmap-like data structure that is optimized for copy-on-write reference counting, part of OSDFS's integrated support for object- and partition-level copy-on-write snapshots.

Block-level metadata management consumes most of the cycles in file system implementations [Gibson97]. By delegating storage management to OSDFS, the Panasas metadata managers have an order of magnitude less work to do than the equivalent SAN file system metadata manager that must track all the blocks in the system.

5.2 File-level Metadata

Above the block layer is the metadata about files. This includes user-visible information such as the owner, size, and modification time, as well as internal information that identifies which objects store the file and how the data is striped across those objects (i.e., the file's *storage map*). Our system stores this file metadata in object attributes on two of the N objects used to store the file's data. The rest of the objects have basic attributes like their individual length and modify times, but the higher-level file system attributes are only stored on the two *attribute-storing* components. Note that the file's length and modify time can be deduced from the corresponding attributes on each object, but for performance we store an explicit file-level version of these attributes that is distinct from the object-level attributes.

Remember the hot-spot problem caused by biasing file creates toward an empty replacement blade? This is because the first two component objects store the file-level attributes, so they see more Set Attributes and Get Attributes traffic than the rest of the components. Files always start out mirrored on these first two attribute-storing components, so the file create bias was creating a metadata hot spot.

File names are implemented in directories similar to traditional UNIX file systems. Directories are special files that store an array of directory entries. A directory entry identifies a file with a tuple of <serviceID, partitionID, objectID>, and also includes two <osdID> fields that are hints about the location of the attribute storing components. The partitionID/objectID is the two-level object numbering scheme of the OSD interface, and we use a partition for each volume. Directories are mirrored (RAID-1) in two objects so that the small write operations associated with directory updates are efficient.

Clients are allowed to read, cache and parse directories, or they can use a Lookup RPC to the metadata manager to translate a name to an <serviceID, partitionID, objectID> tuple and the <osdID> location hints. The serviceID provides a hint about the metadata manager for the file, although clients may be redirected to the

metadata manager that currently controls the file. The `osdID` hint can become out-of-date if reconstruction or active balancing moves an object. If both `osdID` hints fail, the metadata manager has to multicast a `GetAttributes` to the storage nodes in the `BladeSet` to locate an object. The `partitionID` and `objectID` are the same on every storage node that stores a component of the file, so this technique will always work. Once the file is located, the metadata manager automatically updates the stored hints in the directory, allowing future accesses to bypass this step.

File operations may require several object operations. Figure 7 shows the steps used in creating a file. The metadata manager keeps a local journal to record in-progress actions so it can recover from object failures and metadata manager crashes that occur when updating multiple objects. For example, creating a file is fairly complex task that requires updating the parent directory as well as creating the new file. There are 2 Create OSD operations to create the first two components of the file, and 2 Write OSD operations, one to each replica of the parent directory. As a performance optimization, the metadata server also grants the client read and write access to the file and returns the appropriate capabilities to the client as part of the `FileCreate` results. The server makes record of these write capabilities to support error recovery if the client crashes while writing the file. Note that the directory update (step 7) occurs after the reply, so that many directory updates can be batched together. The deferred update is protected by the `op-log` record that gets deleted in step 8 after the successful directory update.

The metadata manager maintains an *op-log* that records the object create and the directory updates that are in progress. This log entry is removed when the operation is complete. If the metadata service crashes and restarts, or a failure event moves the metadata service to a different manager node, then the `op-log` is processed to determine what operations were active at the time of the failure. The metadata manager rolls the operations forward or backward to ensure the object store is consistent. If no reply to the operation has been generated, then the operation is rolled back. If a reply has been generated but pending operations are outstanding (e.g., directory updates), then the operation is rolled forward.

The write capability is stored in a *cap-log* so that when a metadata server starts it knows which of its files are busy. In addition to the “piggybacked” write capability returned by `FileCreate`, the client can also execute a `StartWrite` RPC to obtain a separate write capability.

The `cap-log` entry is removed when the client releases the write cap via an `EndWrite` RPC. If the client reports an error during its I/O, then a *repair log* entry is made and the file is scheduled for repair. Read and write capabilities are cached by the client over multiple system calls, further reducing metadata server traffic.

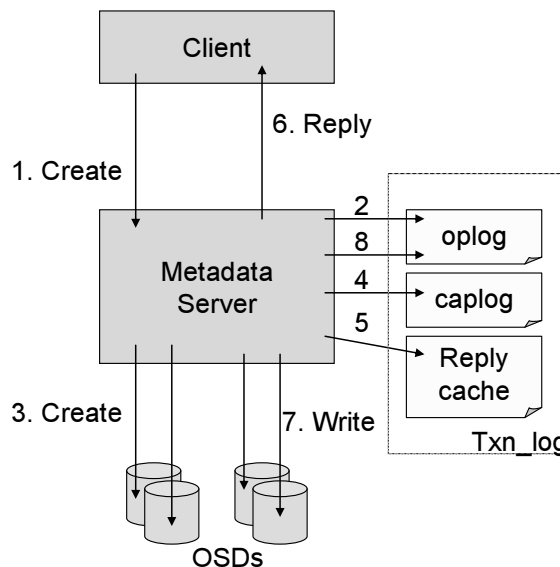


Figure 7: Creating a File

Our log implementation uses a fast (3 μ sec updates) in-memory logging mechanism that is saved to disk on system shutdown, power failure, and software faults, including kernel panics. In fail over configurations this log is synchronously reflected to a remote peer’s memory via a low-latency protocol (90 μ sec update over gigabit Ethernet). Software crashes are usually handled by a restart and recovery from the local logs. If a hardware fault or OS hang disables a `DirectorBlade` module, then its backup takes over and recovers from the log replica.

If the logs are unrecoverable, or there was no backup, then a crash may have left the distributed object store in an inconsistent state (e.g., a partially created file). These inconsistencies are discovered and tolerated during normal operation. In some cases the system can repair the object during the file open operation. In other cases, the system just fences the suspect objects and they can be repaired later via an (offline) file system recovery check facility that sweeps the object store, repairing inconsistencies and rolling in-progress operations forward or back. The optional recovery process runs at about 800 files/sec, and can run in

parallel over multiple Volumes on different DirectorBlade modules.

The combination of fast in-memory logging to battery backed memory, log replication to a remote peer, storing metadata on objects, and an off-line repair process is a robust design point. The in-memory logging is possible because of the integrated UPS that lets us safely shut down our system. The remote backup guards against outright hardware failure of the DirectorBlade module. However, in any complex product, software faults are a hazard. Our metadata manager runs as a user-level application, so it can be killed and restarted transparently and recover based on its logs. The NFS gateway, which uses the kernel-resident client, can be a source of kernel panics. Most of these have been “clean” kernel panics where the system is able to save the in-memory log contents. Finally, if all else fails, we know we can run the recovery process, which will restore the file system to a consistent state. It is comforting to know that the metadata managers can all halt and catch fire, and we can still recover the storage system, since all of the file metadata is resident on the storage nodes themselves.

5.3 File Metadata Performance

We measured metadata performance with the Metarates application [Metarates]. This is an MPI application that coordinates file system accesses from multiple clients. Figure 8 shows Create and Utime performance for fifteen 2.4 GHz Xeon clients against a single DirectorBlade module. The Create operation creates an empty file, and each client works in its own directory. The Utime operation sets the timestamp attributes on a file. Results are presented for the Panasas DirectFLOW protocol (DF) and the NFS protocol that uses the gateway client on the DirectorBlade module. Two different DirectorBlade models were measured: DB-100 and DB-100a. We also measured an NFS server running Linux with a locally attached RAID array.

The Panasas protocol performs better on Utime than NFS because of the way attributes are managed on the clients. The Panasas client can reliably cache attributes because of a callback protocol from the stateful Panasas metadata managers. The NFS clients need to revalidate their cache with a GetAttr operation before they can complete the SetAttr, so they perform twice as many server operations to complete a Utime operation, and hence the client’s throughput is reduced.

In the Linux NFS test the file server does synchronous disk writes during CREATE and SETATTR operations, as required by the stable-storage clause of the NFSv3

standard [Pawlowski94], with a clear performance impact. In the Panasas system, the writes are buffered in protected memory on the StorageBlade modules. Updates are streamed in a log-fashion, and the operation time is decoupled from disk I/O. Our create algorithm is very robust, with journal records resident on two DirectorBlade modules and objects created on two StorageBlade modules before the operation returns to the client. The latency for a single create is about 2 msec.

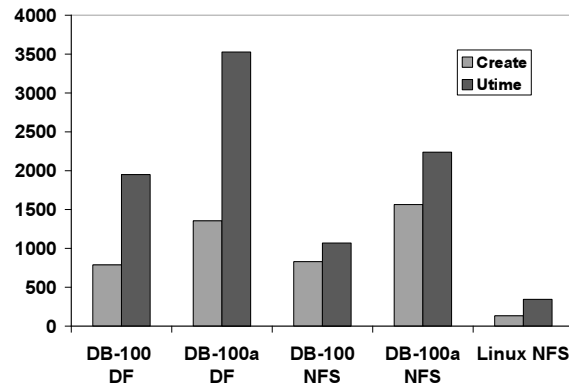


Figure 8: Metarates throughput (ops/sec)

Some NFS servers optimize Create by logging them in NVRAM and returning immediately. This is not done in the Panasas system because clients must be able to write directly to the storage nodes after the Create operation returns from the metadata manager. While we have considered granting create capabilities to clients as an optimization, we have avoided the additional complexity it adds to the protocol. Our metadata manager creates the objects on storage before returning to the client.

Figure 9 shows Create and Utime performance as the number of clients increase. These operations involve I/O operations to the StorageBlade modules. Create does two object creates and two writes to the parent directory. Utime does a set attributes to two objects. The DirectorBlade and shelf correspond to the “DB-100” in the first chart. The 2.4 GHz DirectorBlade module is approaching saturation at 840 creates/sec and 2000 utime/sec with 15 clients.

Figure 10 shows Stat performance. The benchmark has a single client create all the files, followed by all the clients doing their Stat operations. The first client always hits in its cache and consequently gets 45,000 Stat ops/sec. The other clients get 1,200 to 2,000 ops/sec, since their operations involve a server round-trip. The graph charts the Metarates “aggregate

throughput”, which is obtained by multiplying the slowest client times the total number of clients. This metric may seem unfair, but it represents the effective throughput for barrier synchronized MPI applications. At two clients, for example, one ran at 1,777 ops/sec, while the other ran at 45,288 ops/sec, and the aggregate is reported as 3,554 ops/sec.

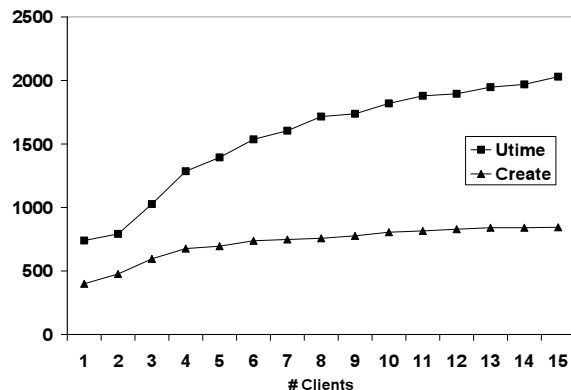


Figure 9: Metarates Create and Utime ops/sec

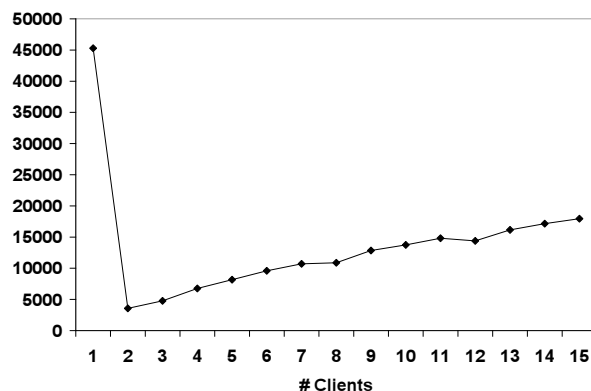


Figure 10: Metarates Stat ops/sec

5.4 System-level Metadata

The final layer of metadata is information about the overall system itself. One possibility would be to store this information in objects and bootstrap the system through a discovery protocol. The most difficult aspect of that approach is reasoning about the fault model. The system must be able to come up and be manageable while it is only partially functional. We chose instead a model with a small replicated set of system managers, each that stores a replica of the system configuration metadata.

Each system manager maintains a local database, outside of the object storage system. We use Berkeley DB to store tables that represent our system model.

The different system manager instances are members of a *replication set* that use Lamport’s part-time parliament (PTP) protocol [Lamport98] to make decisions and update the configuration information. Clusters are configured with one, three, or five system managers so that the voting quorum has an odd number and a network partition will cause a minority of system managers to disable themselves.

System configuration state includes both static state, such as the identity of the blades in the system, as well as dynamic state such as the online/offline state of various services and error conditions associated with different system components. Each state update decision, whether it is updating the admin password or activating a service, involves a voting round and an update round according to the PTP protocol. Database updates are performed within the PTP transactions to keep the databases synchronized. Finally, the system keeps backup copies of the system configuration databases on several other blades to guard against catastrophic loss of every system manager blade.

Blade configuration is pulled from the system managers as part of each blade’s startup sequence. The initial DHCP handshake conveys the addresses of the system managers, and thereafter the local OS on each blade pulls configuration information from the system managers via RPC.

The cluster manager implementation has two layers. The lower level PTP layer manages the voting rounds and ensures that partitioned or newly added system managers will be brought up-to-date with the quorum. The application layer above that uses the voting and update interface to make decisions. Complex system operations may involve several steps, and the system manager has to keep track of its progress so it can tolerate a crash and roll back or roll forward as appropriate.

For example, creating a volume (i.e., a quota-tree) involves file system operations to create a top-level directory, object operations to create an object partition within OSDFS on each StorageBlade module, service operations to activate the appropriate metadata manager, and configuration database operations to reflect the addition of the volume. Recovery is enabled by having two PTP transactions. The initial PTP transaction determines if the volume should be created, and it creates a record about the volume that is marked as incomplete. Then the system manager does all the necessary service activations, file and storage operations. When these all complete, a final PTP transaction is performed to commit the operation. If the

system manager crashes before the final PTP transaction, it will detect the incomplete operation the next time it restarts, and then roll the operation forward or backward.

We measured a simple PTP transaction that updates an entry in the configuration database with a simple test program that performs 1000 of these operations. The cost of an update, which includes the RPC to the system manager, the PTP voting round, and the BDB update to a single table, is around 7 msec on our 2.4 GHz DirectorBlade modules. This cost is dominated by a synchronous disk write in the BDB update. We enabled synchronous disk writes in spite of the UPS so that the system configuration database is highly reliable. The cost doubles to 14 msec when there is a replication set of 2, 3, 4, or 5 members because of an additional table update performed by the PTP implementation. The president of the PTP quorum performs RPCs in parallel to the quorum members, so at these scales of replication there is no performance difference between having 2 or 5 members of the replication set.

Note that there is two or three orders of magnitude difference between the logging performed by the file system metadata manager and the voting transaction performed by the cluster manager. The in-memory log update is 3 microseconds, or 90 microseconds to reflect that across the network. The PTP voting round and BDB database update is 7 to 14 milliseconds. These different mechanisms let us have a very robust cluster management system and a very high performance file system.

6 Related Work

The main file systems that are in production use with high performance compute clusters are the Panasas file system, Lustre [Lustre02], GPFS [Schmuck02], and PVFS2 [PVFS2][Devulapalli07][Yu05]. Cope gives some performance comparisons between Lustre, GPFS, and PVFS2 [Cope06]. Lustre has a similar overall architecture to Panasas, and both systems are based on ideas from the CMU NASD work. Lustre uses relatively larger object storage servers (OSS servers and OST object stores), and can use simple striping across OST but without active capacity balancing or extra parity information. PVS2 also does simple striping across storage nodes without redundancy. GPFS and Lustre rely on block-based RAID controllers to handle disk failure, whereas PVS2 typically uses local file systems on compute nodes.

Clustered NFS systems include Isilon [Isilon], NetApp GX [Klivanski06], and PolyServe[Polyserve]. These

NFS systems have limited scalability because each client must funnel its requests through a single access point, which then forwards requests to the nodes that own the data. The parallel NFS extension [Hildebrand05], which will be part of NFSv4.1, may help these systems overcome this limitation. The Panasas NFS export from the DirectorBlade modules has similar characteristics to these systems.

SAN file systems (e.g., CXFS [Shepard04], [IBRIX], MPSFi [EMC], Oracle's OCFS2 [Fasheh06], GFS2 [GFS2], ADIC StorNext) have non-standard clients, a metadata manager node, and are block oriented. These evolved from single-host file systems by introducing a block manager, or metadata server. Block management is generally a scalability limit because the metadata manager has to track billions of blocks in a system of any size.

GPFS is also a SAN file system, but its distributed lock management scheme and the use of large blocks (256 K to 4 MB) help it scale up to support large clusters. GPFS uses a centralized token manager that delegates fine-grained locks over blocks, inodes, attributes, and directory entries. Lock delegation lets the first client that accesses a resource become the lock manager for it, which spreads out metadata management load. There are workloads, however, that result in a significant amount of traffic between lock owners and the token manager as the system negotiates ownership of locks. Nodes can be both clients and servers, although in large systems there are typically a larger number of client-only nodes, and a subset of nodes that control storage. The services are fault tolerant via fail over protocols and dual-porting of drives.

There are several research projects exploring object storage, including Ceph [Weil06] and Usra Minor [Abd-El-Malek05]. These systems have slightly different object semantics and custom protocols. Usra Minor provides versioning as a basic property of its objects. Ceph uses a hash-based distribution scheme, and its object servers propagate replicas to each other (i.e., there is redundancy but no striping). Lustre uses a distributed lock manager protocol in which clients, OSS, and the metadata manager all participate. The Panasas object model is based on the standard iSCSI/OSD command set that we expect to be part of next generation commodity storage devices.

Striping across data servers for increased bandwidth was evaluated in several research systems. In the Zebra file system [Hartman93], clients would generate a stream of data containing a log of their writes to many files. This log stream was striped across servers, and a

parity component was generated to allow recovery. This approach combines parity information for all the files in the log, which does not allow tuning the per-file RAID configurations. The Cheops parallel file system was layered over NASD and did provide per-file striping, but not per-file RAID [Gibson98]. Isilon stripes files across its storage nodes and uses RAID and Reed Solomon encoding to protect against lost objects. Isilon performance, however, is limited by its NFS interface. The RAIK system [Jukov07] maps a file onto multiple file system and allows striping. This is performed by a stackable file system on the client. However, that work doesn't describe how to handle updates to shared files by multiple clients. Lustre uses simple, RAID-0 striping across object storage servers, and depends on RAID within the server to recover from disk failures, and failover and dual ported drives between servers to handle server failure. Recent versions of NetApp-GX can stripe files and volumes across storage nodes. It also provides facilities for migrating (i.e., copying) file sets between servers to shift load, and it has the ability to configure read-only replicas of a volume for load-sharing, features similar to those introduced by AFS [Howard88]. The Panasas approach to striping is specifically designed to provide performance that scales up to support very large systems with thousands of active clients sharing the same set of files.

Google FS [Ghemawat03] is a user-level file system implemented in application-specific libraries. Google FS uses replication for fault tolerance so it can tolerate loss of a storage node. Clients are responsible for pushing data to replicas, and then notifying the metadata manager when the updates are complete. Google FS provides application-specific append semantics to support concurrent updates. Panasas has fully serialized updates to provide POSIX semantics, and another *concurrent write mode* that optimizes interleaved, strided write patterns to a single file from many concurrent clients.

Of these systems, only Panasas, Isilon, Google, and Ceph use redundant data on different storage nodes to recover from drive failures. The other systems use RAID controllers, or software-RAID within a storage node, or no redundancy at all. The other differentiator is the division of metadata management among nodes. Panasas divides ownership by file trees, allowing multiple metadata managers to manage the overall system. Most other systems have a single metadata manager, including Lustre, IBRIX, and the other SAN file systems. GPFS has a hybrid scheme with a single token manager that can delegate metadata

responsibility. Ceph uses a fine-grained hashing scheme to distribute metadata ownership.

7 Conclusions

This paper has presented the design of the Panasas parallel file system and given several performance measurements that illustrate its scalability. The design uses storage nodes that run an OSDFS object store, and manager nodes that run a file system metadata manager, a cluster manager, and a Panasas file system client that can be re-exported via NFS and CIFS. Scalability comes from the balanced nature of each storage node, which includes disk, CPU, memory, and network bandwidth resources. The Panasas storage cluster is a parallel machine for block allocation because each storage node manages its own drives privately, and it is a parallel machine for RAID rebuild because each manager blade participates in the rebuild of declustered parity groups that are spread across the storage cluster.

Good performance comes from leveraging non-volatile memory to hide latency and protect caches, and the ability to distribute the file server metadata at the block, file, and system level across the storage nodes and manager nodes within the storage cluster. Striping files over objects with per-file RAID protection allows scalable performance for environments with many clients, as well as for the rebuild rates of failed OSDs. We have shown I/O and metadata performance results for systems ranging in size from 11 nodes to 132 nodes in the storage cluster, and from 1 to 100 file system clients.

Appendix I: Experimental Details

This section summarizes the hardware configuration used for the different experiments. Because we have an assorted collection of hardware that spans 3 product generations in our labs, some of the experiments use different hardware.

The StorageBlade module contains a fairly low-powered x86 processor, two disks (250 GB, 500 GB or 750 GB 7200 RPM SATA drives), 512 MB or 2 GB ECC memory, and two GE NICs. The DirectorBlade module has a faster x86 processor, 4 GB ECC memory, two GE NICs, and a small PATA or SATA disk. The specific configuration of each blade model used in our experiments is listed in Table 2. Note that despite having a slower clock speed, the CPU in the DB-100a is about 30% faster for our code than the CPU in the DB-100.

The clients are single CPU, 2.4 GHz or 2.8 GHz Xeon with 1 GE NIC.

Model	CPU	Memory	Disk
SB-160	850 MHz Pentium III	512 MB PC100	2 x 80 GB PATA
SB-500	1.2 GHz Celeron	512 MB PC133	2 x 250 GB SATA
SB-500a	1.5 GHz Celeron M	512 MB DDR2-400	2 x 250 GB SATA
SB-500 XC	1.5 GHz Celeron M	2 GB DDR2-400	2 x 250 GB SATA
1st gen DB	1.6 GHz Xeon	4 GB PC100	40 GB PATA
DB-100	2.4 GHz Xeon	4 GB DDR266	80 GB PATA
DB-100a	1.8 GHz Pentium M	4 GB DDR2-533	80 GB SATA

Table 2: Blade hardware details

The networking environment is 1GE using a Force10 E1200 as a core switch. Client nodes are directly connected to the core switch. Each shelf chassis has a 4 GE trunked connection to the core switch, unless otherwise noted.

Experiment 1. *Scaling clients against different sized storage systems.* Up to 100 clients were used, each running two instances of *iozone*. We used flags `-i 0 -i 1 -e -c -r 64k -s 5g -w` and a `+-m` clients file that put two threads on each client. The StorageBlade modules are SB-160. Each shelf has one first generation DB module and 10 StorageBlade modules, although only a single metadata service is involved in this test.

Experiment 2. *Random I/O vs storage node cache memory and number of clients.* We used *iozone* flags `-i 0 -i 1 -i 2 -i 8 -s 4g` and reported the mixed I/O number. There are 9 SB-500a or SB-500a-XC StorageBlade modules and 2 DB-100 DirectorBlade modules, but only a single DirectorBlade was active. The shelves only had a single GE uplink.

Experiment 3. *Scalable RAID Rebuild.* 10 to 120 SB-160 StorageBlade modules, and 1 to 12 DB DirectorBlade modules. We measured the capacity used on a failed storage node, and divided by the wall clock time reported by the system to complete reconstruction.

Experiment 4. *RAID rebuild vs. stripe width.* The 3+8 system had three DB-100 DirectorBlade modules, and 8 SB-500a-XC StorageBlade modules. The 4+18 had four DB-100a DirectorBlade modules and 18 SB-500a-XC StorageBlade modules in two shelves.

Experiment 5. *Metarates performance.* The two different DirectorBlades were the DB-100 and DB-100a. The NFS server was running Red Hat Enterprise Linux 4 (2.6.9-42.ELsmp) on a 4-way 2.6 GHz Optron 8218 machine with four 10K RPM SAS drives in a RAID-0 configuration and the ext3 file system.

References

Abd-El-Malek05, Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, Jay J. Wylie. *Ursa Minor: Versatile Cluster-based Storage*. Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST '05). San Francisco, CA. December 13-16, 2005.

Cope06. J. Cope, M. Oberg, H.M. Tufo, and M. Woitaszek, *Shared Parallel File Systems in Heterogeneous Linux Multi-Cluster Environments*, proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution.

Devulapalli07, A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, P. Sadayappan. *Integrating Parallel File Systems with Object-Based Storage Devices*, ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2007), Reno, NV, November 2007.

EMC. *EMC Celerra Multi Path File System MPFS/MPFSi*, www.emc.com/products/software/celerra_mpfs/

Fasheh06, Mark Fasheh, *OCFS2: The Oracle Clustered File System, Version 2*, Proceedings of the 2006 Linux Symposium, pp 289-302

GFS2. <http://sourceware.org/cluster/>

Ghemawat03, Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. *The Google File System*, Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003, pp 20-43.

Gibson97. Garth A. Gibson, David Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, Jim Zelenka: *File Server Scaling with Network-Attached Secure Disks*. Proceedings of SIGMETRICS, 1997, Seattle WA. pp 272-284

Gibson98, Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. 1998. *A cost-effective, high-bandwidth storage architecture*. In Proceedings of the Eighth international Conference on Architectural Support For Programming Languages and

Operating Systems (San Jose, California, United States, October 02 - 07, 1998). ASPLOS-VIII

Gobioff97, Gobioff, H., Gibson, G., Tygar, D., *Security for Network Attached Storage Devices*, TR CMU-CS-97-185, Oct 1997.

Hartman93, J. Hartman and J. Ousterhout. *The Zebra Striped Network File System*. Proc. 14-th Symposium on Operating Systems Principles, pages 29-43, December 1993

Hildebrand05, D. Hildebrand, P. Honeyman, *Exporting Storage Systems in a Scalable Manner with pNFS*, MSST 2005.

Hitz94, D. Hitz, J. Lau, and M. Malcolm, *File System Design for an NFS File Server Appliance*, Proceedings of the Winter 1994 USENIX Conference, San Francisco, CA, January 1994, 235-246

Holland92. Mark Holland, Garth Gibson. *Parity declustering for continuous operation in redundant disk arrays*. Proceedings of the 5th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)

Howard88, Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J. "Scale and Performance in a Distributed File System" *ACM Transactions on Computer Systems* Feb. 1988, Vol. 6, No. 1, pp. 51-81.

IBRIX. <http://www.ibrix.com/>

Iozone. <http://www.iozone.org/>

Isilon. <http://www.isilon.com/>

Jukov07, Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok, *RAIF: Redundant Array of Independent Filesystems*, Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)

Klivansky06, Miroslav Klivansky, *Introduction to Data ONTAP GX*, <http://www.netapp.com/library/tr/3468.pdf>, April 2006

Lamport98, L. Lamport. "The Part-Time Parliament." *ACM Transactions on Computer Systems*, Vol. 16 No. 2, 133-169, 1998

Lustre02, Cluster File Systems Inc., *Lustre: A scalable high-performance file system*, lustre.org/documentation.html.

Metarates. www.cisl.ucar.edu/css/software/metarates/

Nagle04 David Nagle, Denis Serenyi, and Abbie Matthews. *The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage*. In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04), November 2004

OSD04, *Object-Based Storage Device Commands (OSD)*, ANSI standard INCITS 400-2004.

Pawlowski94, B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. *NFS version3: Design and implementation*. In Proceedings of the Summer 1994 USENIX Technical Conference, pp 137-151, 1994.

PVFS2, The Parallel Virtual File System, version 2, <http://www.pvfs.org/pvfs2>.

PolyServe, <http://www.polyserve.com>

Rosenblum90, M. Rosenblum, J. Ousterhout, *The LFS Storage Manager*, Proceedings of the 1990 Summer Unix, Anaheim, CA, June 1990, 315-324.

Schmuck02, Frank Schmuck, Roger Haskin, *GPFS: A Shared-Disk File System for Large Computing Clusters*, FAST 02.

Shepard04, Laura Shepard, Eric Eppe, *SGI InfiniteStorage Shared Filesystem CXFS: A High-Performance, Multi-OS Filesystem*, 2004, www.sgi.com/pdfs/2691.pdf.

SNIA SNIA-OSD Working Group, www.snia.org/osd

Weil06, Sage Weil, Scott Brandt, Ethan Miller, Darrell Long, Carlos Maltzahn, *Ceph: A Scalable, High-Performance Distributed File System*, Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06), November 2006.

Welch07. Brent Welch, *Integrated System Models for Reliable Petascale Storage Systems*, Proceedings of the Petascale Data Storage Workshop, Supercomputing '07, Reno NV. November 2007.

Yu05, Weikuan Yu, Shuang Liang, Dhabaleswar Panda, *High Performance Support of Parallel Virtual File System (PVFS2) over Quadrics*, Proceedings of ICS05, Cambridge MA, June 20-22, 2005.

ZFS www.sun.com/software/solaris/ds/zfs.jsp, www.opensolaris.org/os/community/zfs/docs/zfs_last.pdf

TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions

Michael Demmer, Bowei Du, and Eric Brewer
University of California, Berkeley
{demmer,bowei,brewer}@cs.berkeley.edu

Abstract

TierStore is a distributed filesystem that simplifies the development and deployment of applications in challenged network environments, such as those in developing regions. For effective support of bandwidth-constrained and intermittent connectivity, it uses the Delay Tolerant Networking store-and-forward network overlay and a publish/subscribe-based multicast replication protocol. TierStore provides a standard filesystem interface and a single-object coherence approach to conflict resolution which, when augmented with application-specific handlers, is both sufficient for many useful applications and simple to reason about for programmers. In this paper, we show how these properties enable easy adaptation and robust deployment of applications even in highly intermittent networks and demonstrate the flexibility and bandwidth savings of our prototype with initial evaluation results.

1 Introduction

The limited infrastructure in developing regions both hinders the deployment of information technology and magnifies the need for it. In spite of the challenges, a variety of simple information systems have shown real impact on health care, education, commerce and productivity [19, 34]. For example, in Tanzania, data collection related to causes of child deaths led to a reallocation of resources and a 40% reduction in child mortality (from 16% to 9%) [4, 7].

Yet in many places, the options for network connectivity are quite limited. Although cellular networks are growing rapidly, they remain a largely urban and costly phenomenon, and although satellite networks have coverage in most rural areas, they too are extremely expensive [30]. For these and other networking technologies, power problems and coverage gaps cause connectivity to vary over time and location.

To address these challenges, various groups have used novel approaches for connectivity in real-world applications. The Wizzy Digital Courier system [36] distributes educational content among schools in South Africa by delaying dialup access until night time, when rates are cheaper. DakNet [22] provides e-mail and web connectivity by copying data to a USB drive or hard disk and then physically carrying the drive, sometimes via motorcycles. Finally, Ca:sh [1] uses PDAs to gather rural health care data, also relying on physical device transport to overcome the lack of connectivity. These projects demonstrate the value of information distribution applications in developing regions, yet they all essentially started from scratch and thus use ad-hoc solutions with little leverage from previous work.

This combination of demand and obstacles reveals the need for a flexible application framework for “challenged” networks. Broadly speaking, challenged networks lack the ability to support reliable, low-latency, end-to-end communication sessions that typify both the phone network and the Internet. Yet many important applications can still work well despite low data rates and frequent or lengthy disconnections; examples include e-mail, voicemail, data collection, news distribution, e-government, and correspondence education. The challenge lies in implementing systems and protocols to adapt applications to the demands of the environment.

Thus our central goal is to provide a general purpose framework to support applications in challenged networks, with the following key properties: First, to adapt existing applications and develop new ones with minimal effort, the system should offer a familiar and easy-to-use filesystem interface. To deal with intermittent networks, applications must operate unimpeded while disconnected, and easily resolve update conflicts that may occur as a result. Finally, to address the networking challenges, replication protocols need to be able to leverage a range of network transports, as appropriate for particular environments, and efficiently distribute application data.

As we describe in the remainder of this paper, TierStore is a distributed filesystem that offers these properties. Section 2 describes the high-level design of the system, followed by a discussion of related work in Section 3. Section 4 describes the details of how the system operates. Section 5 discusses some applications we have developed to demonstrate flexibility. Section 6 presents an initial evaluation, and we conclude in Section 7.

2 TierStore Design

The goal of TierStore is to provide a distributed filesystem service for applications in bandwidth-constrained and/or intermittent network environments. To achieve these aims, we claim no fundamentally new mechanisms, however we argue that TierStore is a novel synthesis of well-known techniques and most importantly is an effective platform for application deployment.

TierStore uses the Delay Tolerant Networking (DTN) bundle protocol [11, 28] for all inter-node messaging. DTN defines an overlay network architecture for challenged environments that forwards messages among nodes using a variety of transport technologies, including traditional approaches and long-latency “sneakernet” links. Messages may also be buffered in persistent storage during connection outages and/or retransmitted due to a message loss. Using DTN allows TierStore to adapt naturally to a range of network conditions and to use solution(s) most appropriate for a particular environment.

To simplify application development, TierStore implements a standard filesystem interface that can be accessed and updated at multiple nodes in the network. Any modifications to the shared filesystem state are both applied locally and encoded as update messages that are lazily distributed to other nodes in the network. Because nodes may be disconnected for long periods of time, the design favors availability at the potential expense of consistency [12]. This decision is critical to allow applications to function unimpeded in many environments.

The filesystem layer implements traditional NFS-like semantics, including close-to-open consistency, hard and soft links, and standard UNIX group, owner, and permission semantics. As such, many interesting and useful applications can be deployed on a TierStore system without (much) modification, as they often already use the filesystem for communication of shared state between application instances. For example, several implementations of e-mail, log collection, and wiki packages are already written to use the filesystem for shared state and have simple data distribution patterns, and are therefore straightforward to deploy using TierStore. Also, these applications are either already conflict-free in the ways that they interact with shared storage or can be easily made conflict-free with simple extensions.

Based in part on these observations, TierStore implements a *single-object coherence* policy for conflict management, meaning that only concurrent updates to the same file are flagged as conflicts. We have found that this simple model, coupled with application-specific conflict resolution handlers, is both sufficient for many useful applications and easy to reason about for programmers. It is also a natural consequence from offering a filesystem interface, as UNIX filesystems do not naturally expose a mechanism for multiple-file atomic updates.

When conflicts do occur, TierStore exposes all information about the conflicting update through the filesystem interface, allowing either automatic resolution by application-specific scripts or manual intervention by a user. For more complex applications for which single-file coherence is insufficient, the base system is extensible to allow the addition of application-specific meta-objects (discussed in Section 4.12). These objects can be used to group a set of user-visible files that need to be updated atomically into a single TierStore object.

To distribute data efficiently over low-bandwidth network links, TierStore allows the shared data to be partitioned into fine-grained *publications*, currently defined as disjoint subtrees of the filesystem namespace. Nodes can then subscribe to receive updates to only their publications of interest, rather than requiring all shared state to be replicated. This model maps quite naturally to the needs of real applications (*e.g.* users’ mailboxes and folders, portions of web sites, or regional data collection). Finally, TierStore nodes are organized into a multicast-like distribution tree to limit redundant update transmissions over low-bandwidth links.

3 Related Work

Several existing systems offer distributed storage services with varying network assumptions; here we briefly discuss why none fully satisfies our design goals.

One general approach has been to adapt traditional network file systems such as NFS and AFS for use in constrained network environments. For example, the Low-Bandwidth File System (LBFS) [18] implements a modified NFS protocol that significantly reduces the bandwidth consumption requirements. However, LBFS maintains NFS’s focus on consistency rather than availability in the presence of partitions [12], thus even though it addresses the bandwidth problems, it is unsuitable for intermittent connectivity.

Coda [16] extends AFS to support disconnected operation. In Coda, clients register for a subset of files to be “hoarded”, *i.e.* to be available when offline, and modifications made while disconnected are merged with the server state when the client reconnects. Due to its AFS heritage, Coda has a client-server model that imposes re-

strictions on the network topology, so it is not amenable to cases in which there may not be a clear client-server relationship and where intermittency might occur at multiple points in the network. This limits the deployability of Coda in many real-world environments that we target.

Protocols such as rsync [33], Unison [24] and OfflineIMAP [20] can efficiently replicate file or application state for availability while disconnected. These systems provide pairwise synchronization of data between nodes, so they require external ad-hoc mechanisms for multiple-node replication. More fundamentally, in a shared data store that is being updated by multiple parties, no single node has the correct state that should be replicated to all others. Instead, it is the collection of each node's *updates* (additions, modifications, and deletions) that needs to be replicated throughout the network to bring everyone up to date. Capturing these update semantics through pair-wise synchronization of system state is challenging and in some cases impossible.

Bayou [23, 32] uses an epidemic propagation protocol among mobile nodes and has a strong consistency model. When conflicts occur, it will roll back updates and then roll forward to reapply them and resolve conflicts as needed. However, this flexibility and expressiveness comes at a cost: applications need to be rewritten to use the Bayou shared database, and the system assumes that data is fully replicated at every node. It also assumes that rollback is always possible, but in a system with human users, the rollback might require undoing the actions of the users as well. TierStore sacrifices the expressiveness of Bayou's semantic level updates in favor of the simplicity of a state-based system.

PRACTI [2] is a replicated storage system that uses a Bayou-like replication protocol, enhanced with summaries of aggregated metadata to enable multi-object consistency without full database content replication. However, the invalidation-based protocol of PRACTI implies that for strong consistency semantics, it must retrieve invalidated objects on demand. Since these requests may block during network outages, PRACTI either performs poorly in these cases or must fall back to simpler consistency models, thus no longer providing arbitrary consistency. Also, as in the case of Bayou, PRACTI requires a new programming environment with special semantics for reading and writing objects, increasing the burden on the application programmer.

Dynamo [8] implements a key/value data store with a goal of maximum availability during network partitions. It supports reduced consistency and uses many techniques similar to those used in TierStore, such as version vectors for conflict detection and application-specific resolution. However, Dynamo does not offer a full hierarchical namespace, which is needed for some applications, and it is targeted for data center environ-

ments, whereas our design is focused on a more widely distributed topology.

Haggle [29] is a clean-slate design for networking and data distribution targeted for mobile devices. It shares many design characteristics with DTN, including a flexible naming framework, multiple network transports, and late binding of message destinations. The Haggle system model incorporates shared storage between applications and the network, but is oriented around publishing and querying for messages, not providing a replicated storage service. Thus applications must be rewritten to use the Haggle APIs or adapted using network proxies.

Finally, the systems that are closest to TierStore in design are optimistically concurrent peer-to-peer file systems such as Ficus [21] and Rumor [15]. Like TierStore, Ficus implements a shared file system with single file consistency semantics and automatic resolution hooks for update conflicts. However the Ficus log-exchange protocols are not well suited for long latency (*i.e.* sneakernet) links, since they require multiple round trips for synchronization. Also, update conflicts must be resolved before the file becomes available, which can degrade availability in cases where an immediate resolution to the conflict is not possible. In contrast, TierStore allows conflicting partitions to continue to make progress.

Rumor is an external user-level synchronization system that builds upon the Ficus work. It uses Ficus' techniques for conflict resolution and update propagation, thus making it unsuitable in our target environment.

4 TierStore in Detail

This section describes the implementation of TierStore. First we give a brief overview of the various components of TierStore, shown in Figure 1, then we delve into more detail as the section progresses.

4.1 System Components

As discussed above, TierStore implements a standard filesystem abstraction, *i.e.*, a persistent repository for file objects and a hierarchical namespace to organize those files. Applications interface with TierStore using one of two filesystem interfaces, either FUSE [13] (Filesystem in Userspace) or NFS [27]. Typically we use NFS over a loopback mount, though a single TierStore node could export a shared filesystem to a number of users in a well-connected LAN environment over NFS.

File and system data are stored in persistent storage *repositories* that lie at the core of the system. Read access to data passes through the *view resolver* that handles conflicts and presents a self-consistent filesystem to applications. Modifications to the filesystem are encapsulated as updates and forwarded to the *update manager*

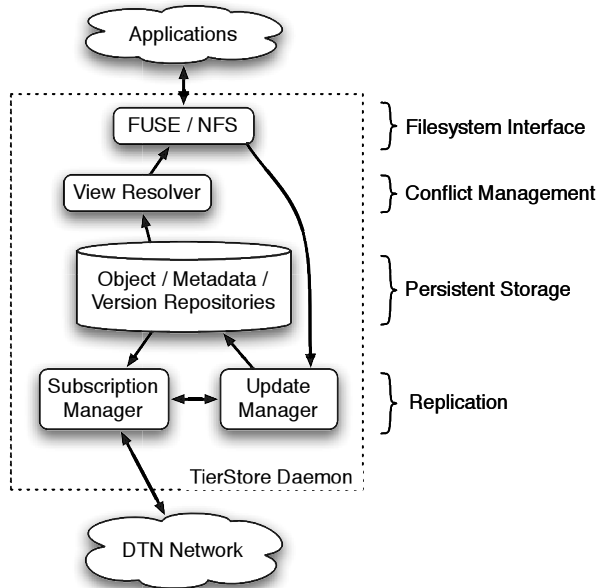


Figure 1: Block diagram showing the major components of the TierStore system. Arrows indicate the flow of information between components.

where they are applied to the persistent repositories and forwarded to the *subscription manager*.

The subscription manager uses the DTN network to distribute updates to and from other nodes. Updates that arrive from the network are forwarded to the update manager where they are processed and applied to the persistent repository in the same way as local modifications.

4.2 Objects, Mappings, and Guides

TierStore objects derive from two basic types: *data objects* are regular files that contain arbitrary user data, except for symbolic links that have a well-specified format. *Containers* implement directories by storing a set of *mappings*: tuples of (*guid*, *name*, *version*, *view*).

A *guid* uniquely identifies an object, independent from its location in the filesystem, akin to an inode number in the UNIX filesystem, though with global scope. Each node in a TierStore deployment is configured with a unique *identity* by an administrator, and *guids* are defined as a tuple (*node*, *time*) of the node identity where an object was created and a strictly increasing local time counter.

The *name* is the user-specified filename in the container. The *version* defines the logical time when the mapping was created in the history of system updates, and the *view* identifies the node the created the mapping (not necessarily the node that originally created the object). Versions and views are discussed further below.

4.3 Versions

Each node increments a local update counter after every new object creation or modification to the filesystem namespace (*i.e.* rename or delete). This counter is used to uniquely identify the particular update in the history of modifications made at the local node, and is persistently serialized to disk to survive reboots.

A collection of update counters from multiple nodes defines a *version vector* and tracks the logical ordering of updates for a file or mapping. As mentioned above, each mapping contains a version vector. Although each version vector conceptually has a column for all nodes in the system, in practice, we only include columns for nodes that have modified a particular mapping or the corresponding object, which is all that is required for the single-object coherence model.

Thus a newly created mapping has only a single entry in its version vector, in the column of the creating node. If a second node were to subsequently update the same mapping, say by renaming the file, then the new mapping's version vector would include the old version in the creating node's column, plus the newly incremented update counter from the second node. Thus the new vector would subsume the old one in the version sequence.

We expect TierStore deployments to be relatively small-scale (at most hundreds of nodes in a single system), which keeps the maximum length of the vectors to a reasonable bound. Furthermore, most of the time, files are updated at an even smaller number of sites, so the size of the version vectors should not be a performance problem. We could, however, adopt techniques similar to those used in Dynamo [8] to truncate old entries from the vector if this were to become a performance limitation.

We also use version vectors to detect missing updates. The subscription manager records a log of the versions for all updates that have been received from the network. Since each modification causes exactly one update counter to be incremented, the subscription manager detects missing updates by looking for holes in the version sequence. Although the DTN network protocols retransmit lost messages to ensure reliable delivery, a fallback repair protocol detects missing updates and can request them from a peer.

4.4 Persistent Repositories

The core of the system has a set of persistent repositories for system state. The *object repository* is implemented using regular UNIX files named with the object *guid*. For data objects, each entry simply stores the contents of the given file. For container objects, each file stores a log of updates to the name/*guid*/*view* tuple set, periodically compressed to truncate redundant entries. We use a log

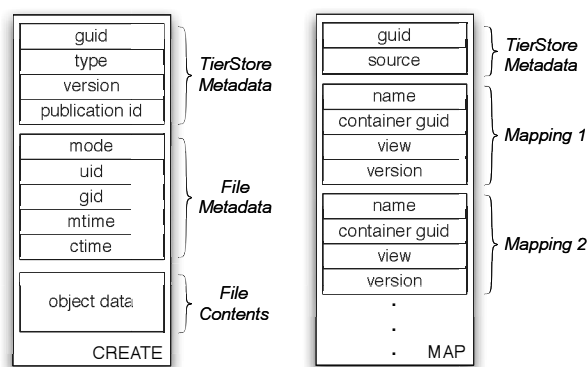


Figure 2: Contents of the core TierStore update messages. CREATE updates add objects to the system; MAP updates bind objects to location(s) in the namespace.

instead of a vector of mappings for better performance on modifications to large directories.

Each object (data and container) has a corresponding entry in the *metadata repository*, also implemented using files named with the object guid. These entries contain the system metadata, *e.g.* user/group/mode/permissions, that are typically stored in an inode. They also contain a vector of all the mappings where the object is located in the filesystem hierarchy.

With this design, mapping state is duplicated in the entries of the metadata table, and in the individual container data files. This is a deliberate design decision: knowing the vector of objects in a container is needed for efficient directory listing and path traversal, while storing the set of mappings for an object is needed to update the object mappings without knowing its current location(s) in the namespace, simplifying the replication protocols.

To deal with the fact that the two repositories might be out of sync after a system crash, we use a write ahead log for all updates. Because the updates are idempotent (as discussed below), we simply replay uncommitted updates after a system crash to ensure that the system state is consistent. We also implement a simple write-through cache for both persistent repositories to improve read performance on frequently accessed files.

4.5 Updates

The filesystem layer translates application operations (*e.g.* write, rename, creat, unlink) into two basic update operations: CREATE and MAP, the format of which is shown in Figure 2. These updates are then applied locally to the persistent repository and distributed over the network to other nodes.

CREATE updates add new objects to the system but do not make them visible in the filesystem namespace. Each

CREATE is a tuple (*object guid, object type, version, publication id, filesystem metadata, object data*). These updates have no dependencies, so they are immediately applied to the persistent database upon reception, and they are idempotent since the binding of a guid to object data never changes (see the next subsection).

MAP updates bind objects into the filesystem namespace. Each MAP update contains the guid of an object and a vector of (*name, container_guid, view, version*) tuples that specify the location(s) where the object should be mapped into the namespace. Although in most cases a file is mapped into only a single location, multiple mappings may be needed to properly handle hard links and some conflicts (described below).

Because TierStore implements a single-object coherence model, MAP updates can be applied as long as a node has previously received CREATE updates for the object and the container(s) where the object is to be mapped. This dependency is easily checked by looking up the relevant guids in the metadata repository and does not depend on other MAP messages having been received. If the necessary CREATE updates have not yet arrived, the MAP update is put into a deferred update queue for later processing when the other updates are received.

An important design decision related to MAP messages is that they contain no indication of any obsolete mapping(s) to *remove* from the namespace. That is because each MAP message implicitly removes all older mappings for the given object and for the given location(s) in the namespace, computed based on the logical version vectors. As described above, the current location(s) of an object can be easily looked up in the metadata repository using the object guid.

Thus, as shown in Figure 3, to process a MAP message, TierStore first looks up the object and container(s) using their respective guids in the metadata repository. If they both exist, then it compares the versions of the mappings in the message with those stored in the repository. If the new message contains more recent mappings, TierStore applies the new set of relevant mappings to the repository. If the message contains old mappings, it is discarded. In case the versions are incomparable (*i.e.* updates occurred simultaneously), then there is a conflict and both conflicting mappings are applied to the repository to be resolved later (see below). Therefore, MAP messages are also idempotent, since any obsolete mappings contained within them are ignored in favor of the more recent ones that are already in the repository.

4.6 Immutable Objects and Deletion

These two message types are sufficient because TierStore objects are immutable. A file modification is implemented by copying an object, applying the change, and

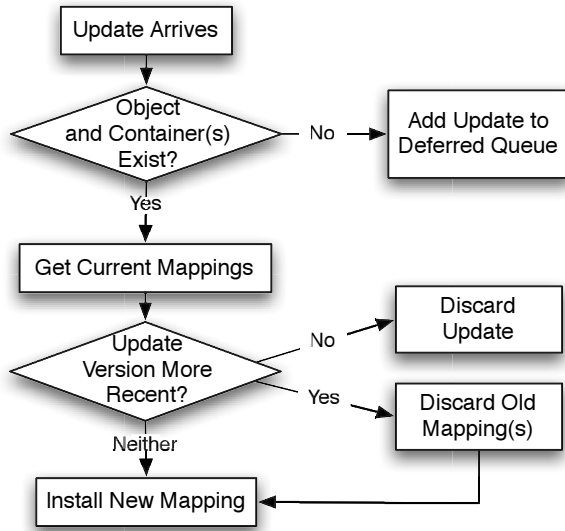


Figure 3: Flowchart of the decision process when applying MAP updates.

installing the modified copy in place of the old one (with a new `CREATE` and `MAP`). Thus the binding of a guid to particular file content is persistent for the life of the system. This model has been used by other systems such as Oceanstore [26], for the advantage that write-write conflicts are handled as name conflicts (two objects being put in the same namespace location), so we can use a single mechanism to handle both types of conflicts.

An obvious disadvantage is the need to distribute whole objects, even for small changes. To address this issue, the filesystem layer only “freezes” an object (*i.e.* issues a `CREATE` and `MAP` update) after the application closes the file, not after each call to `write`. In addition, we plan to integrate other well-known techniques, such as sending deltas of previous versions or encoding the objects as a vector of segments and only sending modified segments (as in LBFS [18]). However, when using these techniques, care would have to be taken to avoid round trips in long-latency environments.

When an object is no longer needed, either because it was explicitly removed with `unlink` or because a new object was mapped into the same location through an edit or `rename`, we do not immediately delete it, but instead we map it into a special trash container. This step is necessary because some other node may have concurrently mapped the object into a different location in the namespace, and we need to hold onto the object to potentially resolve the conflict.

In our current prototype, objects are eventually removed from the trash container after a long interval (*e.g.* multiple days), after which we assume no more updates will arrive to the object. This simple method has been

sufficient in practice, though a more sophisticated distributed garbage collection such as that used in Ficus [21] would be more robust.

4.7 Publications and Subscriptions

One of the key design goals for TierStore is to enable fine-grained sharing of application state. To that end, TierStore applications divide the overall filesystem namespace into disjoint covering subsets called *publications*. Our current implementation defines a publication as a tuple (*container, depth*) that includes any mappings and objects in the subtree that is rooted at the given container, up to the given depth. Any containers that are created at the leaves of this subtree are themselves the root of new publications. By default, new publications have infinite depth; custom-depth publications are created through a special administrative interface.

TierStore nodes then have *subscriptions* to an arbitrary set of publications; once a node is subscribed to a publication, it receives and transmits updates for the objects in that publication among all other subscribed nodes. The *subscription manager* component handles registering and responding to subscription interest and informing the DTN layer to set up forwarding state accordingly. It interacts with the *update manager* to be notified of local updates for distribution and to apply updates received from the network to the data store.

Because nodes can subscribe to an arbitrary set of publications and thus receive a subset of updates to the whole namespace, each publication defines a separate version vector space. In other words, the combination of (*node, publication, update counter*) is unique across the system. This means that a node knows when it has received all updates for a publication when the version vector space is fully packed and has no holes.

To bootstrap the system, all nodes have a default subscription to the special root container “/” with a depth of 1. Thus whenever any node creates an object (or a container) in the root directory, the object is distributed to all other nodes in the system. However, because the root subscription is at depth 1, all containers within the root directory are themselves the root for new publications, so application state can be partitioned.

To subscribe to other publications, users create a symbolic link in a special `/.subscriptions/` directory to point to the root container of a publication. This operation is detected by the *Subscription Manager*, which then sets up the appropriate subscription state. This design allows applications to manage their interest sets without the need for a custom programming interface.

4.8 Update Distribution

To deal with intermittent or long-delay links, the TierStore update protocol is biased heavily towards avoiding round trips. Thus unlike systems based on log exchange (e.g. Bayou, Ficus, or PRACTI), TierStore nodes proactively generate updates and send them to other nodes when local filesystem operations occur.

TierStore integrates with the DTN reference implementation [9] and uses the bundle protocol [28] for all inter-node messaging. The system is designed with minimal demands on the networking stack: simply that all updates for a publication eventually propagate to the subscribed nodes. In particular, TierStore can handle duplicate or out-of-order message arrivals using the versioning mechanisms described above.

This design allows TierStore to take advantage of the intermittency tolerance and multiple transport layer features of DTN. In contrast with systems based on log-exchange, TierStore does not assume there is ever a low-latency bidirectional connection between nodes, so it can be deployed on a wide range of network technologies including sneakernet or broadcast links. Using DTN also naturally enables optimizations such as routing smaller MAP updates over low-latency, but possibly expensive links, while sending large CREATE updates over less expensive but long-latency links, or configuring different publications to use different DTN priorities.

However, for low-bandwidth environments, it is also important that updates be efficiently distributed throughout the network to avoid overwhelming low-capacity links. Despite some research efforts on the topic of multicast in DTNs [38], there currently exists no implementation of a robust multicast routing protocol for DTNs.

Thus in our current implementation, TierStore nodes in a given deployment are configured by hand in a static multicast distribution tree, whereby each node (except the root) has a link to its parent node and to zero or more child nodes. Nodes are added or removed by editing configuration files and restarting the affected nodes. Given the small scale and simple topologies of our current deployments, this manual configuration has been sufficient thus far. However we plan to investigate the topic of a general publish/subscribe network protocol suitable for DTNs in future work.

In this simple scheme, when an update is generated, TierStore forwards it to DTN stack for transmission to the parent and to each child in the distribution tree. DTN queues the update in persistent storage, and ensures reliable delivery through the use of custody transfer and retransmissions. Arriving messages are re-forwarded to the other peers (not back to the sending node) so updates eventually reach all nodes in the system.

4.9 Views and Conflicts

Each mapping contains a *view* that identifies the TierStore node that created the mapping. During normal operation, the notion of views is hidden from the user, however views are important when dealing with conflicts. A conflict occurs when operations are concurrently made at different nodes, resulting in incomparable logical version vectors. In TierStore's single-object coherence model, there are only two types of conflicts: a *name conflict* occurs when two different objects are mapped to the same location by different nodes, while a *location conflict* occurs when the same object is mapped to different locations by different nodes.

Recall that all mappings are tagged with their respective view identifiers, so a container may contain multiple mappings for the same name, but in different views. The job of the *View Resolver* (see Figure 1) is to present a coherent filesystem to the user, in which two files can not appear in the same location, and a single file can not appear in multiple locations. Hard links are an obvious exception to this latter case, in which the user deliberately maps a file in multiple locations, so the view resolver is careful to distinguish hard links from location conflicts.

The default policy to manage conflicts in TierStore appends each conflicting mapping name with `.#X`, where *X* is the identity of the node that generated the conflicting mapping. This approach retains both versions of the conflicted file for the user to access, similar to how CVS handles an update conflict. However, locally generated mappings retain their original name after view resolution and are not modified with the `.#X` suffix. This means that the filesystem structure may differ at different points in the network, yet also that nodes always “see” mappings that they have generated locally, regardless of any conflicting updates that may have occurred at other locations.

Although it is perhaps non-intuitive, we believe this to be an important decision that aids the portability of unmodified applications, since their local file modifications do not “disappear” if another node makes a conflicting update to the file or location. This also means that application state remains self-consistent even in the face of conflicts and most importantly, is sufficient to handle conflicts for many applications. Still, conflicting mappings would persist in the system unless resolved by some user action. Resolution can be manual or automatic; we describe both in the following sections.

4.10 Manual Conflict Resolution

For unstructured data with indeterminate semantics (such as the case of general file sharing), conflicts can be manually resolved by users at any point in the network by using the standard filesystem interface to either remove

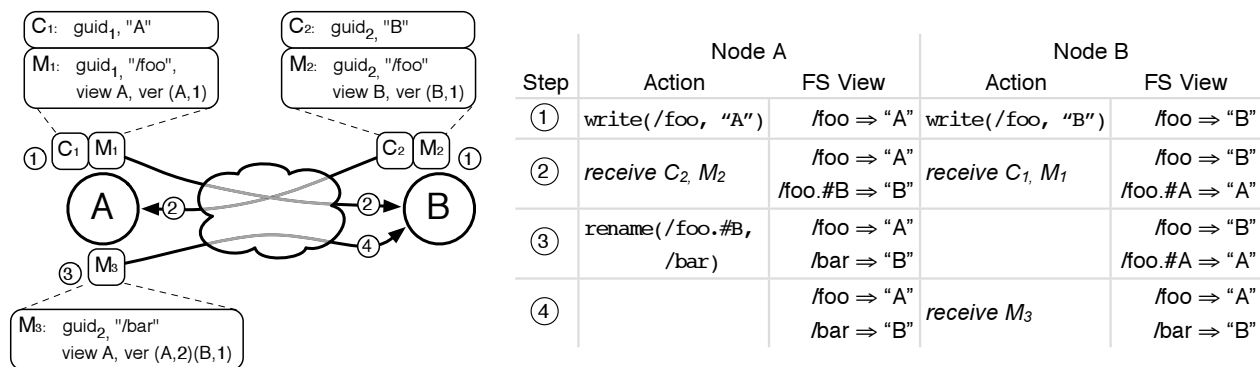


Figure 4: Update sequence demonstrating a name conflict and a user's resolution. Each row in the table at right shows the actions that occur at each node and the nodes' respective views of the filesystem. In step 1, nodes A and B make concurrent writes to the same file `/foo`, generating separate create and mapping updates (C_1 , M_1 , C_2 , and M_2) and applying them locally. In step 2, the updates are exchanged, causing both nodes to display conflicting versions of the file (though in different ways). In step 3, node A resolves the conflict by renaming `/foo.#B` to `/bar`, which generates a new mapping (M_3). Finally, in step 4, M_3 is received at B and the conflict is resolved.

or rename the conflicting mappings. Figure 4 shows an example of how a name conflict is caused, what each filesystem presents to the user at each step, and how the conflict is eventually resolved.

When using the filesystem interface, applications do not necessarily include all the context necessary to infer user intent. Therefore an important policy decision is whether operations should implicitly resolve conflicts or let them linger in the system by default. As in the example shown in Figure 4, once the name conflict occurs in step 2, if the user were to write some new contents to `/foo`, should the new file contents replace both conflicting mappings or just one of them?

The current policy in TierStore is to leave the conflicting mappings in the system until they are explicitly resolved by the user (e.g. by removing the conflicted name), as shown in the example. Although this policy means that conflicting mappings may persist indefinitely if not resolved, it is the most conservative policy and we believe the most intuitive as well, though it may not be appropriate for all environments or applications.

4.11 Automatic Conflict Resolution

Application writers can also configure a custom per-container view resolution routine that is triggered when the system detects a conflict in that container. The interface is a single function with the following signature:

resolve(*local view*, *locations*, *names*) → *resolved*

The operands are as follows: *local view* is the local node identity, *locations* is a list of the mappings that are in conflict with respect to location and *names* is a list of mappings that are in conflict with respect to names.

The function returns *resolved*, which is the list of non-conflicting mappings that should be visible to the user. The only requirements on the implementation of the *resolve* function are that it is deterministic based on its operands and that its output mappings have no conflicts.

In fact, the default view resolver implementation described above is implemented as a *resolve* function that appends the disambiguating suffix for visible filenames. In addition, the *maildir* resolver described in Section 5.1 is another example of a custom view resolver that safely merges mail file status information encoded in the maildir filename. Finally, a built-in view resolver detects identical object contents with conflicting versions and automatically resolves them, rather than presenting them to the user as vacuous conflicts.

An important feature of the *resolve* function is that it creates no new updates, rather it takes the updates that exist and presents a self-consistent file system to the user. This avoids problems in which multiple nodes independently resolve a conflict, yet the resolution updates themselves conflict [14]. Although a side effect of this design is that conflicts may persist in the system indefinitely, they are often eventually cleaned up since modifications to merged files will obsolete the conflicting updates.

4.12 Object Extensions

Another way to extend TierStore with application-specific support is the ability to register custom types for data objects and containers. The current implementation supports C++ object subclassing of the base object and container classes, whereby the default implementations of file and directory access functions can be overridden to provide alternative semantics.

For example, this extension could be used to implement a conflict-free, append-only “log object”. In this case, the log object would in fact be a container, though it would present itself to the user as if it were a normal file. If a user appends a chunk of data to the log (*i.e.* opens the file, seeks to the end, writes the data, and closes the file), the custom type handlers would create a new object for the appended data chunk and add it to the log object container with a unique name. Reading from the log object would simply concatenate all chunks in the container using the partial order of the contained objects’ version vectors, along with some deterministic tiebreaker. In this way multiple locations may concurrently append data to a file without worrying about conflicts, and the system would transparently merge updates into a coherent file.

4.13 Security

Although we have not focused on security features within TierStore itself, security guarantees can be effectively implemented at complementary layers.

Though TierStore nodes are distributed, the system is designed to operate within a single administrative scope, similar to how one would deploy an NFS or CIFS share. In particular, the system is not designed for untrusted, federated sharing in a peer-to-peer manner, but rather to be provisioned in a cooperative network of storage replicas for a particular application or set of applications. Therefore, we assume that configuration of network connections, definition of policies for access control, and provisioning of storage resources are handled via external mechanisms that are most appropriate for a given deployment. In our experience, most organizations that are candidates to use TierStore already follow this model for their system deployments.

For data security and privacy, TierStore supports the standard UNIX file access-control mechanisms for users and groups. For stronger authenticity or confidentiality guarantees, the system can of course store and replicate encrypted files as file contents are not interpreted, except by an application-specific automatic conflict resolver that depends on the file contents.

At the network level, TierStore leverages the recent work in the DTN community on security protocols [31] to protect the routing infrastructure and to provide message security and confidentiality.

4.14 Metadata

Currently, our TierStore prototype handles metadata updates such as `chown`, `chmod`, or `utimes` by applying them only to the local repository. In most cases, the operations occur before updates are generated for an object, so the intended modifications are properly conveyed in the

CREATE message for the given object. However if a metadata update occurs long after an object was created, then the effects of the operation are not known throughout the network until another change is made to the file contents.

Because the applications we have used so far do not depend on propagation of metadata, this shortcoming has not been an issue in practice. However, we plan to add a new META update message to contain the modified metadata as well as a new metadata version vector in each object. A separate version vector space is preferable to allow metadata operations to proceed in parallel with mapping operations and to not trigger false conflicts. Conflicting metadata updates would be resolved by a deterministic policy (*e.g.* take the intersection of permission bits, later modification time, etc).

5 TierStore Applications

In this section we describe the initial set of applications we have adapted to use TierStore, showing how the simple filesystem interface and conflict model allows us to leverage existing implementations extensively.

5.1 E-mail Access

One of the original applications that motivated the development of TierStore was e-mail, as it is the most popular and fastest-growing application in developing regions. In prior work, we found that commonly used web-mail interfaces are inefficient for congested and intermittent networks [10]. These results, plus the desire to extend the reach of e-mail applications to places without a direct connection to the Internet, motivate the development of an improved mechanism for e-mail access.

It is important to distinguish between e-mail delivery and e-mail access. In the case of e-mail delivery, one simply has to route messages to the appropriate (single) destination endpoint, perhaps using storage within the network to handle temporary transmission failures. Existing protocols such as SMTP or a similar DTN-based variant are adequate for this task.

For e-mail access, users need to receive and send messages, modify message state, organize mail into folders, and delete messages, all while potentially disconnected, and perhaps at different locations, and existing access protocols like IMAP or POP require clients to make a TCP connection to a central mail server. Although this model works well for good-quality networks, in challenged environments users may not be able to get or send new mail if the network happens to be unavailable or is too expensive at the time when they access their data.

In the TierStore model, all e-mail state is stored in the filesystem and replicated to any nodes in the system where a user is likely to access their mail. An off-

the-shelf IMAP server (*e.g.* courier [6]) runs at each of these endpoints and uses the shared TierStore filesystem to store users' mailboxes and folders. Each user's mail data is grouped into a separate publication, and via an administrative interface, users can instruct the TierStore daemon to subscribe to their mailbox.

We use the `maildir` [3] format for mailboxes, which was designed to provide safe mailbox access without needing file locks, even over NFS. In `maildir`, each message is a uniquely named independent file, so when a mailbox is replicated using TierStore, most operations are trivially conflict free. For example, a disconnected user may modify existing message state or move messages to other mailboxes while new messages are simultaneously arriving without conflict.

However, it is possible for conflicts to occur in the case of user mobility. For example, if a user accesses mail at one location and then moves to another location before all updates have fully propagated, then the message state flags (*i.e.* passed, replied, seen, draft, etc) may be out of sync on the two systems. In `maildir`, these flags are encoded as characters appended to the message filename. Thus if one update sets a certain state, while another concurrently sets a different state, the TierStore system will detect a location conflict on the message object.

To best handle this case, we wrote a simple conflict resolver that computes the union of all the state flags for a message, and presents the unified name through the filesystem interface. In this way, the fact that there was an underlying conflict in the TierStore object hierarchy is never exposed to the application, and the state is safely resolved. Any subsequent state modifications would then subsume both conflicting mappings and clean up the underlying (yet invisible) conflict.

5.2 Content Distribution

TierStore is a natural platform to support content distribution. At the publisher node, an administrator can arbitrarily manipulate files in a shared repository, divided into publications by content type. Replicas would be configured with read-only access to the publication to ensure that the application is trivially conflict-free (since all modifications happen at one location). The distributed content can then be served by a standard web server or simply accessed directly through the filesystem.

As we discuss further in Section 6.2, using TierStore for content distribution is more efficient and easier to administer than traditional approaches such as `rsync` [33]. In particular, TierStore's support for multicast distribution provides an efficient delivery mechanism for many networks that would require ad-hoc scripting to achieve with point-to-point synchronization solutions. Also, the use of the DTN overlay network enables easier integra-

tion of transport technologies such as satellite broadcast [17] or sneakernet and opens up potential optimizations such as sending some content with a higher priority.

5.3 Offline Web Access

Although systems for offline web browsing have existed for some time, most operate under the assumption that the client node will have periodic direct Internet access, *i.e.* will be "online", to download content that can later be served when "offline". However, for poorly connected sites or those with no direct connection at all, TierStore can support a more efficient model, where selected web sites are crawled periodically at a well-connected location, and the cached content is then replicated.

Implementing this model in TierStore turned out to be quite simple. We configured the `wwwoffle` proxy [37] to use TierStore as its filesystem for its cache directories. By running web crawls at a well-connected site through the proxy, all downloaded objects are put in the `wwwoffle` data store, and TierStore replicates them to other nodes. Because `wwwoffle` uses files for internal state, if a remote user requests a URL that is not in cache, `wwwoffle` records the request in a file within TierStore. This request is eventually replicated to a well-connected node that will crawl the requested URL, again storing the results in the replicated data store.

We ran an early deployment of TierStore and `wwwoffle` to accelerate web access in the Community Information Center kiosks in rural Cambodia [5]. For this deployment, the goal was to enable accelerated web access to selected web sites, but still allow direct access to the rest of the Internet. Therefore, we configured the `wwwoffle` servers at remote nodes to always use the cached copy of the selected sites, but to never cache data for other sites, and at a well-connected node, we periodically crawled the selected sites. Since the sites changed much less frequently than they were viewed, the use of TierStore, even on a continuously connected (but slow) network link, was able to accelerate the access.

5.4 Data Collection

Data collection represents a general class of applications that TierStore can support well. The basic data flow model for these applications involves generating log records or collecting survey samples at poorly connected edge nodes and replicating these samples to a well-connected site.

Although at a fundamental level, it may be sufficient to use a messaging interface such as e-mail, SMS, or DTN bundling for this application, the TierStore design offers a number of key advantages. In many cases, the local node wants or needs to have access to the data after it

	CREATE	READ	WRITE	GETDIR	STAT	RENAME
Local	1.72 (0.04)	16.75 (0.08)	1.61 (0.01)	7.39 (0.01)	3.00 (0.01)	27.00 (0.2)
FUSE	3.88 (0.1)	20.31 (0.08)	1.90 (0.8)	8.46 (0.01)	3.18 (0.005)	30.04 (0.07)
NFS	11.69 (0.09)	19.75 (0.06)	42.56 (0.6)	8.17 (0.01)	3.76 (0.01)	36.03 (0.03)
TierStore	7.13 (0.06)	21.54 (0.2)	2.75 (0.3)	15.38 (0.01)	3.19 (0.01)	38.39 (0.05)

Table 1: Microbenchmarks for various file system operations for local Ext3, loopback-mounted NFS, passthrough FUSE layer and TierStore. Runtime is in seconds averaged over five runs, with the standard error in parenthesis.

has been collected, thus some form of local storage is necessary anyway. Also, there may be multiple destinations for the data; many situations exist in which field workers operate from a rural office that is then connected to a larger urban headquarters, and the pub/sub system of replication allows nodes at all these locations to register data interest in any number of sample sets.

Furthermore, certain data collection applications can benefit greatly from fine-grained control over the units of data replication. For example, consider a census or medical survey being conducted on portable devices such as PDAs or cell phones by a number of field workers. Although replicating all collected data samples to every device will likely overwhelm the limited storage resources on the devices, it would be easy to set up publications such that the list of *which* samples had been collected would be replicated to each device to avoid duplicates.

Finally, this application is trivially conflict free. Each device or user can be given a distinct directory for samples, and/or the files used for the samples themselves can be named uniquely in common directories.

5.5 Wiki Collaboration

Group collaboration applications such as online Wiki sites or portals generally involve a set of web scripts that manipulate page revisions and inter-page references in a back-end infrastructure. The subset of common wiki software that uses simple files (instead of SQL databases) is generally quite easy to adapt to TierStore.

For example, PmWiki [25] stores each Wiki page as an individual file in the configured `wiki.d` directory. The files each contain a custom revision format that records the history of updates to each file. By configuring the `wiki.d` directory to be inside of TierStore, multiple nodes can update the same shared site when potentially disconnected.

Of course, simultaneous edits to the same wiki page at different locations can easily result in conflicts. In this case, it is actually safe to do nothing at all to resolve the conflicts, since at any location, the wiki would still be in a self-consistent state. However, users would no longer easily see each other's updates (since one of the conflicting versions would be renamed as described in

Section 4.9), limiting the utility of the application.

Resolving these types of conflicts is also straightforward. PmWiki (like many wiki packages) contains built in support for managing simultaneous edits to the same page by presenting a user with diff output and asking for confirmation before committing the changes. Thus the conflict resolver simply renames the conflicting files in such a way that the web scripts prompt the user to manually resolve the conflict at a later time.

6 Evaluation

In this section we present some initial evaluation results to demonstrate the viability of TierStore as a platform. First we run some microbenchmarks to demonstrate that the TierStore filesystem interface has competitive performance to traditional filesystems. Then we describe experiments where we show the efficacy of TierStore for content distribution on a simulation of a challenged network. Finally we discuss ongoing deployments of TierStore in real-world scenarios.

6.1 Microbenchmarks

This set of experiments compares TierStore's filesystem interface with three other systems: Local is the Linux Ext3 file system; NFS is a loopback mount of an NFS server running in user mode; FUSE is a `fuse_xmp` instance that simply passes file system operations through the user space daemon to the local file system. All of the benchmarks were run on a 1.8 GHz Pentium 4 with 1 GB of memory and a 40GB 7200 RPM EIDE disk, running Debian 4.0 and the 2.6.18 Linux kernel.

For each filesystem, we ran several benchmark tests: CREATE creates 10,000 sequentially named empty files. READ performs 10,000,000 16 kilobyte `read()` calls at random offsets of a one megabyte file. WRITE performs 10,000,000 16k `write()` calls to append to a file; the file was truncated to 0 bytes after every 1,000 writes. GETDIR issues 1,000 `getdir()` requests on a directory containing 800 files. STAT issues 1,000,000 `stat` calls to a single file. Finally, RENAME performs 10,000 `rename()` operations to change a single file back and forth between two filenames. Table 1 summarizes the results of our ex-

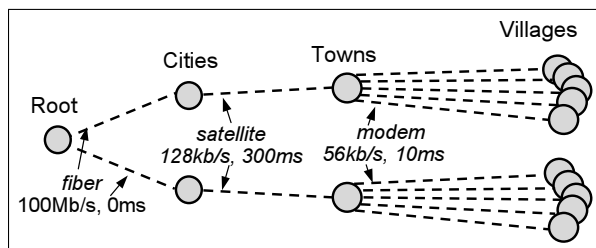


Figure 5: Network model for the emulab experiments.

periments. Run times are measured in seconds, averaged over five runs, with the standard error in parentheses.

The goal of these experiments is to show that existing applications, written with standard filesystem performance in mind, can be deployed on TierStore without worrying about performance barriers. These results support this goal, as in many cases the TierStore system performance is as good as traditional systems. The cases where the TierStore performance is worse are due to some inefficiencies in how we interact with FUSE and the lack of optimizations on the backend database.

6.2 Multi-node Distribution

In another set of experiments, we used the Emulab [35] environment to evaluate the TierStore replication protocol on a challenged network similar to those found in developing regions.

To simulate this target environment, we set up a network topology consisting of a single root node, with a well-connected “fiber” link (100 Mbps, 0 ms delay) to two nodes in other “cities”. We then connect each of these city nodes over a “satellite” link (128 kbps, 300 ms delay) to an additional node in a “village”. In turn, each village connects to five local computers over “dialup” links (56 kbps, 10 ms delay). Figure 5 shows the network model for this experiment.

To model the fact that real-world network links are both bandwidth-constrained and intermittent, we ran a periodic process to randomly add and remove firewall rules that block transfer traffic on the simulated dialup links. Specifically, the process ran through each link once per second, comparing a random variable to a threshold parameter chosen to achieve the desired downtime percentage, and turning on the firewall (blocking the link) if the threshold was met. It then re-opened a blocked link after waiting 20 seconds to ensure that all transport connections closed.

We ran experiments to evaluate TierStore’s performance for electronic distribution of educational content, comparing TierStore to rsync [33]. We then measured the time and bandwidth required to transfer 7MB of multimedia data from the root node to the ten edge nodes.

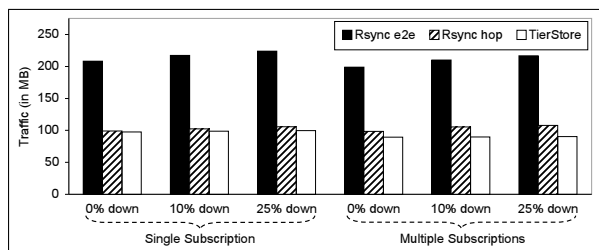


Figure 6: Total network traffic consumed when synchronizing educational content on an Emulab simulation of a challenged network in developing regions. As the network outage increases, the performance of TierStore relative to both end to end and hop by hop rsync improves.

We ran two sets of experiments, one in which all data is replicated to all nodes (single subscription), and another in which portions of the data are distributed to different subsets of the edge nodes (multiple subscriptions). The results from our experiments are shown in Figure 6.

We compared TierStore to rsync in two configurations. The end-to-end model (rsync e2e) is the typical use case for rsync, in which separate rsync processes are run from the root node to each of the edge nodes until all the data is transferred. As can be seen from the graphs, however, this model has quite poor performance, as a large amount of duplicate data must be transferred over the constrained links, resulting in more total traffic and a corresponding increase in the amount of time to transfer (not shown). As a result, TierStore uses less than half of the bandwidth of rsync in all cases. This result, although unsurprising, demonstrates the value of the multicast-like distribution model of TierStore to avoid sending unnecessary traffic over a constrained network link.

To offer a fairer comparison, we also ran rsync in a hop-by-hop mode, in which each node distributed content to its downstream neighbor. In this case, rsync performs much better, as there is less redundant transfer of data over the constrained link. Still, TierStore can adapt better to intermittent network conditions as the outage percentage increases. This is primarily because rsync has no easy way to detect when the distribution is complete, so it must repeatedly exchange state even if there is no new data to transmit. This distinction demonstrates the benefits of the push-based distribution model of TierStore as compared to state exchange when running over bandwidth-constrained or intermittent networks.

Finally, although this latter mode of rsync essentially duplicates the multicast-like distribution model of TierStore, rsync is significantly more complicated to administer. In TierStore, edge nodes simply register their interest for portions of the content, and the multicast replication occurs transparently, with the DTN stack taking care of re-starting transport connections when they

break. In contrast, multicast distribution with rsync required end-to-end application-specific synchronization processes, configured with aggressive retry loops at each hop in the network, making sure to avoid re-distributing partially transferred files multiple times, which was both tedious and error prone.

6.3 Ongoing Deployments

We are currently working on several TierStore deployments in developing countries. One such project is supporting community radio stations in Guinea Bissau, a small West African country characterized by a large number of islands and poor infrastructure. For many of the islands' residents, the main source of information comes from the small radio stations that produce and broadcast local content.

TierStore is being used to distribute recordings from these stations throughout the country to help bridge the communication barriers among islands. Because of the poor infrastructure, connecting these stations is challenging, requiring solutions like intermittent long-distance WiFi links or sneakernet approaches like carrying USB drives on small boats, both of which can be used transparently by the DTN transport layer.

The project is using an existing content management system to manage the radio programs over a web interface. This system proved to be straightforward to integrate with TierStore, again because it was already designed to use the filesystem to store application state, and replicating this state was an easy way to distribute the data. We are encouraged by early successes with the integration and are currently in the process of preparing a deployment for some time in the next several months.

7 Conclusions

In this paper we described TierStore, a distributed filesystem for challenged networks in developing regions. Our approach stems from three core beliefs: the first is that dealing with intermittent connectivity is a necessary part of deploying robust applications in developing regions, thus network solutions like DTN are critical. Second, a replicated filesystem is a natural interface for applications and can greatly reduce the burden of adapting applications to the intermittent environment. Finally, a focus on conflict avoidance and a single-object coherence model is both sufficient for many useful applications and also eases the challenge of programming. Our initial results are encouraging, and we hope to gain additional insights through deployment experiences.

Acknowledgements

Thanks to anonymous reviewers and to our shepherd, Margo Seltzer, for providing insightful feedback on earlier versions of this paper.

Thanks also to Pauline Tweedie, the Asia Foundation, Samnang Yuth Vireak, Bunhoun Tan, and the other operators and staff of the Cambodia CIC project for providing us with access to their networks and help with our prototype deployment of TierStore.

This material is based upon work supported by the National Science Foundation under Grant Number 0326582 and by the Defense Advanced Research Projects Agency under Grant Number 1275918.

Availability

TierStore is freely available open-source software. Please contact the authors to obtain a copy.

References

- [1] Vishwanath Anantraman, Tarjei Mikkelsen, Reshma Khilnani, Vikram S Kumar, Rao Machiraju, Alex Pentland, and Lucila Ohno-Machado. Handheld computers for rural healthcare, experiences in a large scale implementation. In *Proc. of the 2nd Development by Design Workshop (DYD02)*, 2002.
- [2] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jian-dan Zheng. PRACTI replication. In *Proc. of the 3rd ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.
- [3] D.J. Bernstein. Using maildir format. <http://cr.yp.to/proto/maildir.html>.
- [4] Eric Brewer, Michael Demmer, Bowei Du, Melissa Ho, Matthew Kam, Sergiu Nedevschi, Joyojeet Pal, Rabin Patra, Sonesh Surana, and Kevin Fall. The case for technology in developing regions. *IEEE Computer*, 38(6):25–38, June 2005.
- [5] Cambodia Community Information Centers. <http://www.cambodiaticic.info>.
- [6] Courier Mail Server. <http://www.courier-mta.org>.
- [7] Don de Savigny, Harun Kasale, Conrad Mbuya, and Graham Reid. In *Focus: Fixing Health Systems*. International Research Development Centre, 2004.
- [8] Giuseppe DeCandia, Deniz Hastorun, Madan Jambani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, 2007.
- [9] Delay Tolerant Networking Reference Implementation. <http://www.dtnrg.org/wiki/Code>.

- [10] Bowei Du, Michael Demmer, and Eric Brewer. Analysis of WWW Traffic in Cambodia and Ghana. In *Proc. of the 15th international conference on World Wide Web (WWW)*, 2006.
- [11] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proc. of the ACM Symposium on Communications Architectures & Protocols (SIGCOMM)*, 2003.
- [12] Armando Fox and Eric Brewer. Harvest, yield and scalable tolerant systems. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems (HotOS)*, 1999.
- [13] Fuse: Filesystem in Userspace. <http://fuse.sf.net>.
- [14] Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce, and Alan Schmitt. Agreeing to Agree: Conflict Resolution for Optimistically Replicated Data. In *Proc. of the International Symposium on Distributed Computing (DISC)*, 2006.
- [15] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Proc. of ACM International Conference on Conceptual Modeling (ER) Workshop on Mobile Data Access*, pages 254–265, 1998.
- [16] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.
- [17] Dirk Kutscher, Janico Greifenberg, and Kevin Loos. Scalable DTN Distribution over Uni-Directional Links. In *Proc. of the SIGCOMM Workshop on Networked Systems in Developing Regions Workshop (NSDR)*, August 2007.
- [18] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A Low-Bandwidth Network File System. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.
- [19] Sergiu Nedeveschi, Joyojeet Pal, Rabin Patra, and Eric Brewer. A Multi-disciplinary Approach to Studying Village Internet Kiosk Initiatives: The case of Akshaya. In *Proc. of Policy Options and Models for Bridging Digital Divides*, March 2005.
- [20] OfflineIMAP. <http://software.complete.org/offlineimap>.
- [21] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software—Practice and Experience*, 28(2):155–180, February 1998.
- [22] Alex (Sandy) Pentland, Richard Fletcher, and Amir Hason. DakNet: Rethinking Connectivity in Developing Nations. *IEEE Computer*, 37(1):78–83, January 2004.
- [23] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.
- [24] Benjamin C. Pierce and Jerome Vouillon. What’s in Union? A Formal Specification and Reference Implementation of a File Synchronizer. Technical Report MS-CIS-03-36, Univ. of Pennsylvania, 2004.
- [25] PmWiki. <http://www.pmwiki.org/>.
- [26] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore Prototype. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, March 2003.
- [27] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. of the USENIX Summer Technical Conference*, Portland, OR, 1985.
- [28] Keith Scott and Scott Burleigh. RFC 5050: Bundle Protocol Specification, 2007.
- [29] Jing Su, James Scott, Pan Hui, Eben Upton, Meng How Lim, Christophe Diot, Jon Crowcroft, Ashvin Goel, and Eyal de Lara. Haggle: Clean-slate Networking for Mobile Devices. Technical Report UCAM-CL-TR-680, University of Cambridge, Computer Laboratory, January 2007.
- [30] Lakshminarayanan Subramanian, Sonesh Surana, Rabin Patra, Sergiu Nedeveschi, Melissa Ho, Eric Brewer, and Anmol Sheth. Rethinking Wireless in the Developing World. In *Proc. of the 5th Workshop on Hot Topics in Networks (HotNets)*, November 2006.
- [31] Susan Symington, Stephen Farrell, and Howard Weiss. Bundle Security Protocol Specification. Internet Draft draft-irtf-dtnrg-bundle-security-04.txt, September 2007. Work in Progress.
- [32] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.
- [33] A. Tridgell and P. MacKerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National Univ., June 1996.
- [34] Voxiva. <http://www.voxiva.com/>.
- [35] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.
- [36] Wizzy Digital Courier. <http://www.wizzy.org.za/>.
- [37] WWWOFFLE: World Wide Web Offline Explorer. <http://www.gedanken.demon.co.uk/wwwoffle/>.
- [38] Wenrui Zhao, Mostafa Ammar, and Ellen Zegura. Multicasting in Delay Tolerant Networks: Semantic Models and Routing Algorithms. In *Proc. of the ACM SIGCOMM Workshop on Delay-Tolerant Networking (WDTN)*, 2005.

All Internet URLs in citations are valid as of January 2008.

On Multi-level Exclusive Caching: Offline Optimality and

Why promotions are better than demotions.

Binny S. Gill

IBM Almaden Research Center

binnyg@us.ibm.com

Abstract—Multi-level cache hierarchies have become very common; however, most cache management policies result in duplicating the same data redundantly on multiple levels. The state-of-the-art exclusive caching techniques used to mitigate this wastage in multi-level cache hierarchies are the DEMOTE technique and its variants. While these achieve good hit ratios, they suffer from significant I/O and computational overheads, making them unsuitable for deployment in real-life systems.

We propose a dramatically better performing alternative called PROMOTE, which provides exclusive caching in multi-level cache hierarchies without demotions or any of the overheads inherent in DEMOTE. PROMOTE uses an adaptive probabilistic filtering technique to decide which pages to “promote” to caches closer to the application. While both DEMOTE and PROMOTE provide the same aggregate hit ratios, PROMOTE achieves more hits in the highest cache levels leading to better response times. When inter-cache bandwidth is limited, PROMOTE convincingly outperforms DEMOTE by being 2x more efficient in bandwidth usage. For example, in a trace from a real-life scenario, PROMOTE provided an average response time of 3.42ms as compared to 5.05ms for DEMOTE on a two-level hierarchy of LRU caches, and 5.93ms as compared to 7.57ms on a three-level cache hierarchy.

We also discover theoretical bounds for optimal multi-level cache performance. We devise two offline policies, called OPT-UB and OPT-LB, that provably serve as upper and lower bounds on the theoretically optimal performance of multi-level cache hierarchies. For a series of experiments on a wide gamut of traces and cache sizes OPT-UB and OPT-LB ran within 2.18% and 2.83% of each other for two-cache and three-cache hierarchies, respectively. These close bounds will help evaluate algorithms and guide future improvements in the field of multi-level caching.

I. INTRODUCTION

Very rarely does data reach its consumer without traveling through a cache. The performance benefit of a cache is significant, and even though caches are much more expensive than mass storage media like disks, nearly all data servers, web servers, databases, and in fact most computing devices are equipped with a cache. In the last several decades numerous read caching algorithms have been devised (for example, LRU[10], CLOCK[8], FBR[29], 2Q[20], LRFU[21], LIRS[18], MQ[36], ARC[25] and SARC[14]). Most of the work, however, has focused on the case when a single significant layer of cache separates the data producer and the data consumer. In practice, however, data travels

through multiple layers of cache before reaching an application. It has been observed that single-level cache replacement policies perform very poorly when used in multi-level caches [26].

We propose a simple and universal probabilistic technique, called PROMOTE, that adapts any single-level read caching algorithm into an algorithm that allows a multi-level read cache hierarchy to perform as effectively as a single cache of the aggregate size. Unlike previous algorithms, this novel algorithm imposes negligible computational and I/O overheads. As another key contribution to the field of multi-level caching, we provide, for the first time, techniques to compute very close bounds for the notoriously elusive offline optimal performance of multi-level caches.

A. The Problem with Inclusion

One of the earliest examples of multi-level caches arose in the context of a processor [30], [28]. The multiple layers of cache were named L1, L2, L3, etc., with L1 (highest cache) being the closest to the processor, the smallest in size, and the fastest in response time. For efficient cache coherency, systems enforced the inclusion property [1], which mandated that the higher caches be a subset of the lower caches. Other than in cases where L2 was only marginally larger than L1 [22], the performance penalty of this redundancy of content between the layers was not of much concern.

In stark contrast, in the hierarchy of caches formed by multiple computing systems connected to each other, inclusion, sometimes partial, is not by design and has been found to be detrimental to performance [26], [13]. The redundancy of data between the cache levels is most severe when the caches are of comparable sizes. A request is always serviced from the closest cache to the client that has the data, while further copies of the data in lower caches are not useful.

B. Working towards Exclusion

In cache hierarchies through which pages traverse fixed paths from the data source to the application, exclusivity of all caches is highly desirable. In *multi-path* cache hierarchies, where pages can get accessed via

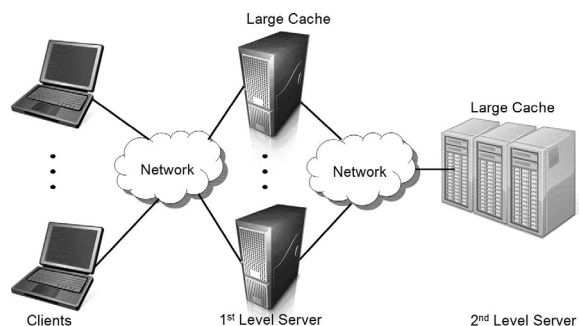


Fig. 1. Cache hierarchies that benefit from exclusive caching.

multiple paths, exclusive caching should be applied to those portions of the cache hierarchy which do not have a large workload overlap between the various paths.

Figure 1 shows a common scenario for which various exclusive caching algorithms have been proposed ([36], [33], [19], [12]). It is fairly common to find cache hierarchies formed by a first level of application servers (database servers, web proxy servers, web content delivery servers, storage virtualization servers) which act as clients for backend storage servers, while both are equipped with significant and comparable caches [33]. In some cases, it may also be possible to include the end clients to form an exclusive cache hierarchy of three or more levels of cache.

A naïve way of enforcing the exclusion property would be to associate different caches with different logical addresses. Obviously, this will not be able to gather frequently hit pages in the topmost caches and the average response time for some workloads might end up worse than the case when the caches are not exclusive at all. Succinctly, the challenge of multi-level exclusive caching is: “*Exclusivity achieved efficiently with most hits at the highest cache levels*”.

C. Exclusivity via Smart Lower Caches

It has been known that the Least Frequently Used ([13], [32]) algorithm performs better at second-level caches than the traditional LRU algorithm. A more sophisticated second-level cache management algorithm is the MQ algorithm [36] which maintains multiple lists geared to capture frequently accessed pages with long reuse intervals. However, it is not studied for more than two levels of cache and also cannot achieve complete exclusivity of the caches where desirable.

A more recent algorithm, X-RAY [2], constructs in the RAID controller cache an approximate image of the contents of the filesystem cache by monitoring the meta-data updates, which allows for better cache replacement decisions and exclusivity. Such gray-box approaches,

however, are domain-specific and not easily applicable to more than two levels of cache.

A similar approach is to use a Client Cache Tracking (CCT) table [6] in the lower cache to simulate the contents of the higher cache. This allows the lower cache to proactively reload the evicted pages from the storage media. The extra cost of these requests, however, may overburden the storage media resulting in high read response times.

D. Exclusivity via Smart Higher Caches

Chen et al. [7] propose an algorithm called AC_{CA} in which a client (higher cache) simulates the contents of the server (lower cache), and uses that knowledge to preferably evict those pages which are also present in the server. This is difficult to do in multi-client scenarios or where the server behavior is complex or proprietary.

E. Exclusivity via Multi-level Collaboration

When multiple layers of cache can be modified to communicate with each other, most of the guesswork of simulating cache contents can be avoided. Even though extensions are required to the communication protocol, this class has proven to be the most promising in terms of performance and ease of implementation.

1) *Application controlled*: The Unified and Level-aware Caching (ULC) algorithm [19] controls a cache hierarchy from the highest level by issuing RETRIEVE and DEMOTE commands to lower caches causing them to move blocks up and down the hierarchy, respectively. The highest cache level (application client) has to keep track of the contents of all the caches below, which entails complexity in the client’s implementation.

2) *Application hints*: KARMA [12] is aimed at applications such as databases that can provide hints for placement decisions in all cache levels. Such solutions are application-specific and do not lend themselves to general applicability in multi-level caches. This brings us to DEMOTE, which is the most popular general-purpose collaborative multi-level caching technique.

F. The DEMOTE Technique

The DEMOTE technique [33], or equivalently the *Global* technique [35], shown in Figure 2, can be applied to many general purpose single-level caching policies (like, LRU, MQ, ARC, etc) to create multi-level versions that achieve exclusivity of cache contents. As with any exclusive caching scheme, DEMOTE should only be used in scenarios that benefit from exclusive caching [33], [27].

G. The Problems with DEMOTE

While the DEMOTE technique strives to improve the aggregate hit ratio over the non-exclusive variant, the overall performance might in fact suffer because

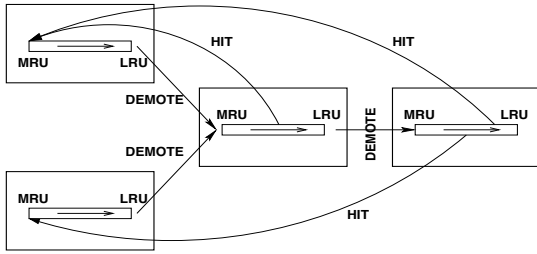


Fig. 2. The DEMOTE technique. When a client is about to eject a clean block from its cache, it sends the clean block to the lower cache using the DEMOTE operation. The lower cache puts the demoted block into its cache, ejecting another block if necessary to make space. Hits in any list are sent to the MRU end of the appropriate highest cache.

of the cost of the DEMOTE operation, including: (i) network traffic to send evicted pages to lower caches, and (ii) processor cycles consumed to prepare, send and receive demoted pages. This has thwarted the practical deployment of DEMOTE in real systems.

In cases where inter-cache bandwidth is tight, or the workload throughput is high, the average response time suffers in spite of a higher hit ratio. This happens as reads stall until demotions (sometimes in multiple levels) can create space for the new page. Further, an eviction, which used to be a trivial operation, now becomes an expensive operation almost equivalent to a write request to the lower cache. This leads to further degradation of performance. Add to this the concern that for workloads that do not exhibit temporal locality, like purely sequential (or purely random), all demotions are futile and we end up wasting critical network resources when most needed.

Eviction-based cache replacement [6] was proposed to alleviate the network bandwidth wastage. A list of evicted pages is sent periodically to the lower cache which reloads them from the disks into its cache. We have observed for many real systems and benchmarks these extra reads increase the average miss penalty, diminishing or defeating the benefits of any increase in hit ratio that such exclusive caching can provide. There have been other attempts to partially mitigate the cost of demotions [34] or the cost of the extra I/Os on the disks [6] by grouping them and using ‘idle time’. In our opinion, idle time should never be considered free as it can be used by other algorithms like prefetching to improve read performance. Sophisticated enterprise-class systems which run round the clock strive hard to minimize idle-time. ULC and KARMA do reduce the number of low reuse pages that enter the higher caches and thereby minimizing the bandwidth wastage. However, ULC’s complexity and KARMA’s dependence on application hints make them less generally applicable.

H. Our Contributions

We present two major results:

Bounds for Optimal Performance: In the study of caching algorithms, it is invaluable to know the offline optimal performance that a multi-level cache can deliver. While Belady’s MIN [4] (an offline algorithm) is considered optimal for single-level caches, it cannot be applied to multi-level cache scenarios, where, apart from the hit ratio, the location of hits is also extremely important. Our first contribution provides insight into the optimal offline performance of multi-level caches.

We provide policies called OPT-UB and OPT-LB that provably serve as upper and lower bounds for the optimal offline performance for multi-level caches along both average response time and inter-cache bandwidth usage metrics.

Through a series of experiments on a wide gamut of traces, cache sizes and configurations, we demonstrate that OPT-UB and OPT-LB are very close bounds on the optimal average response time, running on an average, within 2.18% and 2.83% of each other for all the tested two-cache and three-cache hierarchies, respectively. Even for more complex hierarchies, the bounds remain close at about 10% of each other. This novel result enables us to estimate for the first time, the performance gap between the current state-of-the-art algorithms and the offline optimal for multi-level caches.

PROMOTE Technique: As another fundamental contribution to the field of caching, we propose a simple and significantly better alternative to the DEMOTE technique, called PROMOTE, which provides exclusive caching without demotions, application hints, or any of the overheads inherent in DEMOTE. PROMOTE uses a probabilistic filtering technique to “promote” pages to higher caches on a read. Not only do we show that PROMOTE is applicable to a wider range of algorithms and cache hierarchies, it is on an average, 2x more efficient than DEMOTE requiring only half the inter-cache bandwidth between the various cache levels.

In a wide variety of experiments, while both techniques achieved the same aggregate hit ratio, PROMOTE provided 13.0% and 37.5% more hits in the highest cache than DEMOTE when the techniques were applied to LRU and ARC [25] algorithms, respectively, leading to better average response times even when we allow DEMOTE unlimited inter-cache bandwidth and free demotions. In limited bandwidth scenarios, PROMOTE convincingly outperforms DEMOTE. For example, in a trace from a real-life scenario, PROMOTE provided an average response time of 3.21ms as compared to 5.43ms for DEMOTE on a two-level hierarchy of ARC caches, and 5.61ms as compared to 8.04ms on a three-level cache hierarchy.

II. OFFLINE OPTIMAL

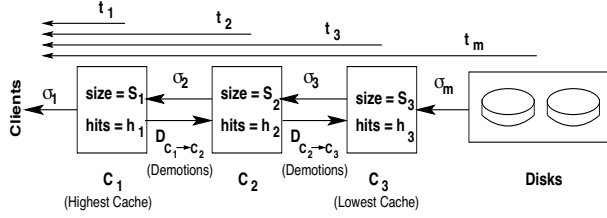


Fig. 3. A multi-level single-path cache hierarchy.

A. Quest for Offline Optimality

The offline optimal performance, which is the best possible performance given full knowledge of the future requests, is a critical guide for cache algorithm development. The offline optimal algorithm for single-level caches is the classic Belady's MIN or OPT [4] which simply evicts the page with the furthest re-reference time. For four decades, the quest for a similar algorithm for multi-level caches has remained unfulfilled. Hit ratio, the performance metric that served us so well in single-level caches, loses its fidelity to performance in multi-level caches, where the benefit of a hit depends on which level it occurred in (e.g. two hits on a higher cache could be better than three hits on a lower cache.)

The average read response time is a much better performance metric, but is complicated in the presence of demotions and/or eviction-based reloading from disks. The inter-cache bandwidth usage is another important metric since many scenarios are bottle-necked by network resources [33]. It does appear that different policies would be optimal depending on which performance metric we use.

We now prove universal bounds for the single-path scenario as depicted in Figure 3 that simultaneously apply to both the average response time and inter-cache bandwidth metrics. The bounds apply to all algorithms irrespective of whether demotions, inclusion, or hints are used. Later, we explain extensions to compute the bounds for the multi-path scenarios as well.

C_i	Cache at level i , $i = 1, 2, \dots, n$
S_i	Size of the cache C_i
h_i	Number of hits at C_i
t_i	Avg. round-trip response time from C_i
t_m	Avg. round-trip response time from disks
σ_i	Sequence of reads seen by C_i
σ_m	Sequence of reads (misses) seen by disks
$D_{c_i \rightarrow c_{i+1}}$	Number of demotions from C_i to C_{i+1}

From Figure 3 and definitions above, the total response time is

$$\text{totalRespTime} = \sum_{i=1}^n h_i \cdot t_i + |\sigma_m| \cdot t_m \quad (1)$$

and the inter-cache traffic between caches C_i and C_{i+1} is

$$\text{interCacheTraffic}_{c_i \rightarrow c_{i+1}} = |\sigma_{i+1}| + |D_{c_i \rightarrow c_{i+1}}| \quad (2)$$

We define $\text{hitOPT}(\sigma, S)$ to be the number of hits produced by Belady's offline OPT policy on the request sequence σ and a single-level cache of size S .

B. Optimal Upper Bound: OPT-UB

We define a conceptual policy OPT-UB that serves as an *upper bound on performance*, implying that no policy can perform better than this bound in terms of either average response time or inter-cache bandwidth usage by achieving better (lower) values for either metric. OPT-UB is a bound, not on the performance of a particular cache, but on the aggregate performance of the cache hierarchy.

Let OPT-UB be a policy that for a request sequence σ , exhibits for each cache C_i , h_i number of hits, while requiring no demotions or reloads from disks, where,

$$h_i = \text{hitOPT}(\sigma, \sum_{j=1}^i S_j) - \text{hitOPT}(\sigma, \sum_{j=1}^{i-1} S_j) \quad (3)$$

Note that we intend to compute a theoretical upper bound which is necessarily achievable.

Lemma II.1 *No policy can have more hits, up to any cache level, than OPT-UB. More precisely, OPT-UB maximizes $\sum_{i=1}^k h_k, \forall k \leq n$.*

Proof: By Eqn. 3 the aggregate hits for the set of caches C_1, \dots, C_k is

$$\text{aggrHits}_k = \sum_{i=1}^k h_i = \text{hitOPT}(\sigma, \sum_{i=1}^k S_i) \quad (4)$$

By the definition of hitOPT , this is the same as that obtained by Belady's OPT on a cache of the aggregate size. Since Belady's OPT is known to deliver the maximum possible hit ratio, aggrHits_k is maximized for all $k \leq n$. ■

Theorem II.1 *No policy can have a lower total inter-cache traffic than OPT-UB.*

Proof: Inter-cache traffic is the sum of misses and demotions between two adjacent caches (by Eqn. 2). Since, OPT-UB is defined to have no demotions and maximizes the aggregate hits at all levels of the cache (Lemma II.1), no other policy can have lower inter-cache traffic than OPT-UB. ■

Theorem II.2 *No policy can have a lower average response time than OPT-UB.*

Proof: We prove by contradiction. Let a better performing policy achieve a lower average response time (equivalently, lower total response time) for a workload than OPT-UB, by providing h'_i hits at each corresponding cache C_i , and m' overall misses, as compared to h_i hits and m misses for OPT-UB.

$$\begin{aligned}
& \therefore \sum_{i=1}^n h'_i \cdot t_i + m' \cdot t_m < \sum_{i=1}^n h_i \cdot t_i + m \cdot t_m \\
& \Rightarrow \sum_{i=1}^n h'_i \cdot (t_i - t_m) + (m' + \sum_{i=1}^n h'_i) \cdot t_m \\
& \quad < \sum_{i=1}^n h_i \cdot (t_i - t_m) + (m + \sum_{i=1}^n h_i) \cdot t_m \\
& \Rightarrow^{(a)} \sum_{i=1}^n h'_i \cdot (t_m - t_i) > \sum_{i=1}^n h_i \cdot (t_m - t_i) \\
& \Rightarrow \sum_{i=1}^n h'_i \cdot (t_n - t_i + t_m - t_n) > \\
& \quad \sum_{i=1}^n h_i \cdot (t_n - t_i + t_m - t_n) \\
& \Rightarrow^{(b)} \sum_{i=1}^n h'_i \cdot (t_n - t_i) > \sum_{i=1}^n h_i \cdot (t_n - t_i) \\
& \quad + \left(\sum_{i=1}^n h_i - \sum_{i=1}^n h'_i \right) \cdot (t_m - t_n) \\
& \Rightarrow^{(c)} \sum_{i=1}^n h'_i \cdot (t_n - t_i) > \sum_{i=1}^n h_i \cdot (t_n - t_i) \\
& \Rightarrow^{(d)} \sum_{i=1}^{n-1} h'_i \cdot (t_n - t_i) > \sum_{i=1}^{n-1} h_i \cdot (t_n - t_i)
\end{aligned}$$

(a) follows as the sum of all hits and misses is the same for both policies ($|\sigma_1|$) and the second term on both sides of the inequality can be removed. The second term on the right hand side of (b) is non-negative because $t_m > t_n$ and by Lemma II.1, no policy can have more aggregate hits (up to any cache level) than OPT-UB. (c) follows by removing the non-negative second term in inequality (b). (d) follows as the n^{th} term in the summation is zero. Note that between step (a) and step (d), the superscript of the summation has dropped from n to $n-1$. Steps (a) through (d) can be repeated until $n=2$ (as, for all i , $t_i > t_{(i-1)}$). We will then arrive at $h'_1 \cdot (t_2 - t_1) > h_1 \cdot (t_2 - t_1)$. As $t_2 > t_1$, it implies that $h'_1 > h_1$, which contradicts Lemma II.1, which states that OPT-UB maximizes $\sum_{i=1}^k h_k, \forall k \leq n$ (including $k=1$). ■

C. Optimal Lower Bound: OPT-LB

We now introduce a very simple and practical offline algorithm called OPT-LB, which provides a very close

lower bound on optimal multi-level caching performance. A better performing policy will have to demonstrate either lower average response time or lower inter-cache bandwidth usage. OPT-LB is the best known offline multi-level caching policy that we are aware of.

The basic idea is to apply Belady's OPT algorithm in a cascaded fashion. We start with the highest cache level, C_1 , and apply OPT to the request sequence σ_1 , assuming a single cache scenario. We note the misses from C_1 with their timestamps and prepare another request sequence, σ_2 , for the lower cache C_2 . We repeat the same process at C_2 , in turn generating σ_3 . This is performed for each level and we keep a count of hits obtained at every level.

In other words, $h_i = \text{hitOPT}(\sigma_i, S_i)$ and $\sigma_{i+1} = \text{traceOPT}(\sigma_i, S_i)$, where traceOPT is a trace of the misses when OPT is applied to the given request stream and cache size.

Once each level has learned its σ_i , all cache levels can operate together replicating the result in real-time. Since this can be done practically, this policy by definition serves as the lower bound for the offline optimal along any performance metric. Note that OPT-LB does not require any demotions or reloads from disks. Even though OPT-LB does not guarantee exclusivity of caches, we experimentally confirm that OPT-LB is indeed a very close lower bound for offline optimals for both average response time and inter-cache bandwidth usage in multi-level caches.

D. Bounds for Multi-path Hierarchies

It is simple to extend OPT-UB and OPT-LB for any complex cache hierarchy. While it is fairly common to accept the use of traces for multi-client caching experiments ([33], [19]), the results are accurate only in cases where the relative order of requests from various clients can be assumed fixed. The same holds true for OPT-UB and OPT-LB, which are valid bounds for multi-path hierarchies if we can assume that the relative order of requests from various clients is fixed. Note that there is no such caveat in single client scenarios, where trace-based analysis is accurate.

We extend OPT-UB as follows: we start with determining the maximum hit ratio obtainable at each cache at the highest level by applying Belady's OPT. Similarly, we determine the maximum aggregate hit ratio obtainable in each two-level subtree starting at the highest levels. We subtract the hit ratio of the highest level caches to obtain the hit ratio for the second-level caches. We do this until we have hit ratio values for all cache levels, using which, we arrive at the OPT-UB average response time value. This is a simple generalization of the single-path approach.

ADAPTING PROBPROMOTE:

```

float hintFreq = 0.05;
float sizeRatio =  $\sum_{i=1}^{k-1} S_i / \sum_{i=1}^k S_i$ ; (at level  $k$ )
float probPromote = sizeRatio;
struct adaptHint {
    time_t life; // life of the cache
    float occupancy; // fraction of cache occupied by  $T_2$ 
                    // -only required by PROMOTE-ARC
} higherHint; // hint from higher cache

Every cache.life * hintFreq
1: Prepare and send adaptHint to lower cache

On receiving adaptHint  $h$ 
2:  $higherHint = h$ ;
3: Every alternate time (to observe the impact
   of the previous adaptation): adjustIfNeeded();

adjustIfNeeded()
4: static float prev = 0;
5: float curr = adaptRatio(); /* algo-specific */
6: float  $f = (2 * curr - 1)$ ; /*  $f = 0$  is the target */
7: if ( $(f > 0 \ \&\&$ 
8:      $prev - curr < hintFreq * (prev - 0.5)) \ ||$ 
9:      $(f < 0 \ \&\&$ 
10:     $curr - prev < hintFreq * (0.5 - prev)))$ 
11:    probPromote +=
12:     $(1 - probPromote) * probPromote * f$ ;
13:    if ( $probPromote > sizeRatio$ )
14:        probPromote = sizeRatio;
15:    endif
16: endif
17: prev = curr;

shouldPromoteUpwards()
18: if ( $HighestCache \ ||$ 
19:     $probPromote < randomBetween0and1()$ )
20:    return false;
21: endif
22: return true;

```

Fig. 4. The common methods used by PROMOTE to adapt $probPromote$ within every cache by leveraging the hint protocol.

We extend OPT-LB for multi-path hierarchies by merging traces (according to timestamps) emerging from higher caches before applying them to the lower cache. We present a couple of illustrative examples towards the end of the paper (Figures 14 and 15).

III. THE PROMOTE TECHNIQUE

The goal of the PROMOTE technique is to provide exclusive caching that performs better than DEMOTE, while at the same time, requires no demotions, reloads from disks, or any computationally intense operation. Each cache independently and probabilistically decides whether to keep the data exclusively as it is passed along to the application. The probability of holding the data or promoting the data upwards is adaptively determined.

PROMOTE-LRU:

```

adaptRatio() /* return a value between 0 and 1 */
23: return  $higherHint.life / (cache.life + higherHint.life)$ 

```

```

On receiving from higher cache (readReq  $addr$ )
24: page  $p = lookupCache(addr)$ ;
25: if ( $p$ ) /* hit */
26:      $promoteHint = shouldPromoteUpwards()$ ;
27:     if ( $promoteHint$ )
28:         remove page  $p$  from cache
29:     endif
30:     send to higher cache ( $p, promoteHint$ )
31: else /* miss */
32:     send to lower cache ( $addr$ )
33: endif

```

```

On receiving from lower cache (page  $p$ , bool  $promoteHint$ )
34: if ( $promoteHint$ )
35:      $promoteHint = shouldPromoteUpwards()$ ;
36:     if ( $!promoteHint$ )
37:         create page  $p$  in cache
38:     endif
39: endif
40: send to higher cache ( $p, promoteHint$ )

```

PROMOTE-ARC:

```

adaptRatio() /* returns a value between 0 and 1 */
41: float higher =  $higherHint.occupancy / higherHint.life$ ;
42: float self =  $T_2.occupancy / T_2.life$ ;
43: return  $self / (self + higher)$ ;

```

```

On receiving from higher cache (readReq  $addr$ , bool  $T_2hint$ )
44: page  $p = lookupCache(addr)$ ;
45: if ( $p \ || \ addr$  found in history)
46:      $T_2hint = true$ ;
47:     remove  $addr$  from history (if present)
48: endif
49: if ( $p$ ) /* hit */
50:      $promoteHint = shouldPromoteUpwards()$ ;
51:     if ( $promoteHint$ )
52:         remove page  $p$  from cache
53:     endif
54:     send to higher cache ( $p, promoteHint, T_2hint$ )
55: else /* miss */
56:     send to lower cache ( $addr, T_2hint$ )
57: endif

```

```

On receiving from lower cache (page  $p$ ,
                               bool  $promoteHint$ , bool  $T_2hint$ )

```

```

58: if ( $promoteHint$ )
59:      $promoteHint = shouldPromoteUpwards()$ ;
60:     if ( $!promoteHint$ )
61:         if ( $T_2hint$ )
62:             create page  $p$  in  $T_2$ 
63:         else
64:             create page  $p$  in  $T_1$ 
65:         endif
66:     endif
67: endif
68: send to higher cache ( $p, promoteHint, T_2hint$ )

```

Fig. 5. The PROMOTE augmentations to ARC and LRU algorithms. These enhancements are apart from the regular mechanisms (not shown) inherent in these algorithms.

We present the PROMOTE algorithm in Figures 4 and 5.

A. Achieving Exclusivity

The PROMOTE technique ensures that only one cache claims ownership of a page on its way to the client. On subsequent accesses to the page, the page can either stay in the same cache or be promoted upwards. As in DEMOTE, only when a page is accessed via different paths in a multi-path hierarchy can a page appear in multiple locations (which is better than enforcing absolute exclusivity).

B. Using *promoteHint* with a READ Reply

While DEMOTE uses an additional bandwidth-intensive operation (DEMOTE: similar to a page WRITE), PROMOTE uses a boolean bit, called *promoteHint*, that is passed along with every READ reply, and helps to decide which cache should own the page. The *promoteHint* is set to true when a READ reply is first formed (cache hit or disk read) and set to false in the READ reply when some cache decides to keep (own) it. A READ reply with a false *promoteHint* implies that a lower cache has already decided to own the page and the page should not be cached at the higher levels. When a cache receives a READ reply with a true *promoteHint*, the cache gets to decide whether to keep (own) the page locally or “promote” it upwards. At each cache level PROMOTE maintains a separate *probPromote* value, which is the probability with which that cache will promote a page upwards. If a cache decides to own the page (Lines 18-22), it changes the *promoteHint* to false in the reply and maintains a copy of the page locally. In all other cases, a READ reply is simply forwarded to the higher cache with the *promoteHint* value unchanged, and any local copy of the data is deleted from the cache. The highest cache has *probPromote* = 0, implying that it always owns a page that it receives with a true *promoteHint*. A page received with a false *promoteHint*, implying that the page is already owned by a lower cache, is merely discarded upon returning it to the application.

C. Promoting Hits Upwards

Pages that incur repeated hits in PROMOTE are highly likely to migrate upwards in the hierarchy as they are subjected repeatedly to the *probPromote* probability of migrating upwards. The more the number of hits incurred by a page the more it can climb in the hierarchy. The most hit pages soon begin to accumulate in the topmost level.

Note that while DEMOTE uses inter-cache bandwidth for pages on their way up towards the application and also their way down, PROMOTE saves bandwidth by moving pages only in one direction.

D. Adapting *probPromote*

Each cache maintains and adapts a separate *probPromote* value. The reader will appreciate that the lower the *probPromote* value at a cache, the lesser is the rate at which new pages will enter the caches above it. Thus, by changing the value of *probPromote*, a lower cache can influence the influx rate of new pages at the higher caches. The fundamental idea in the PROMOTE technique is to use this leverage to create a situation where the “usefulness” of pages evicted from the various caches are roughly the same (if possible). This is different from DEMOTE, where pages evicted from the higher cache are more useful (and hence need to be demoted) than the pages evicted from the lower cache.

To facilitate the adaptation, PROMOTE requires a very simple interface to periodically receive adaptation information like the *cache life* (the timestamp difference between the MRU and LRU pages in the cache) of the next higher cache(s). At regular intervals, each cache, other than the lowest, sends the hint to the next lower cache (Lines 1-3), using which, the *adaptRatio* is computed (Lines 23, 41-43). The goal is to adapt *probPromote* in such a way that the *adaptRatio* approaches 0.5 (a value that implies that the usefulness of pages evicted from the higher cache is the same as of those from the lower cache). If the higher cache has a larger life (*adaptRatio* > 0.5), *probPromote* is increased, else it is decreased. Since there is a lag between the adaptation in *probPromote* and its impact on the *adaptRatio*, the recomputation of *probPromote* (Lines 4-17) is done only on alternate times the hint is received (Line 3). Further, if the previous adaptation of *probPromote* is found to have reduced the separation of *adaptRatio* and 0.5 by a reasonable fraction (Lines 7-10), then no further adaptation is done. To avoid any runaway adaptation, *probPromote* needs to be carefully adapted so that the adaptation is proportional to the difference between *adaptRatio* and 0.5 and also is slower when close to extreme values of 0 and 1 (Lines 11-12). To start off in a fair fashion, *probPromote* is initialized according to the sizes of the caches. Since the higher caches usually demonstrate higher hit rates than the lower caches, we forbid *probPromote* to go beyond the ratio thus determined (Lines 13-14).

Let us examine some examples of PROMOTE in existing cache management policies:

1) PROMOTE-LRU : As shown in Figure 6, LRU is implemented at each cache level, augmented by the PROMOTE protocol. The dynamic adaptation of *probPromote* at each level, results in equalizing the cache lives and it can be shown that the cache hierarchy achieves a hit ratio equal to that of a single cache of the aggregate size. The same is true, if instead of the

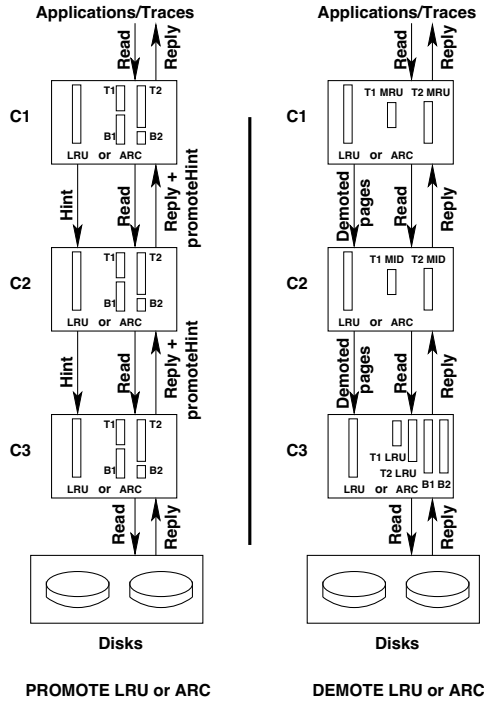


Fig. 6. Communication protocols for the PROMOTE technique (left panel) and the DEMOTE technique (right panel). Although shown in the same diagram, either LRU or ARC data-structures are used consistently across all levels.

cache lives we choose to equalize the marginal utility of the caches. The marginal utility can be computed by measuring the hit rate on a fixed number of pages in the LRU ends of the caches [14]. To avoid the extra complexity, we do not use the marginal utility approach in this paper.

2) PROMOTE-ARC : First, we summarize the ARC algorithm [25]: ARC maintains c pages in the cache and c history pages. LRU list T_1 contains pages that have been seen only once recently, and T_2 contains the pages that have been seen at least twice recently. The addresses of the pages recently evicted from T_1 and T_2 are stored in B_1 and B_2 , respectively. $|T_1| + |T_2| \leq c$ is enforced, while the relative sizes of the lists are adapted according to the relative rates of hits in the corresponding history lists B_1 and B_2 .

For simplicity, consider a single-path hierarchy of ARC caches (Figure 6). Theoretically, the caches could pass the marginal utility of the T_1 and T_2 lists to lower caches which could dynamically adapt *probPromote* at each cache level to equalize the utility across the hierarchy. However, it can be challenging to adapt *probPromote* in a stable way for both T_1 and T_2 lists at each level. We found a simple policy that works very well for ARC. For traffic destined for T_1 lists, *probPromote* at each cache C_k is set to a fixed value

$\sum_{i=1}^{k-1} |C_i| / \sum_{i=1}^k |C_i|$. Instead of the cache life, the *life/occupancy* of the T_2 list is passed to lower caches, where *occupancy* is the fraction of the cache occupied by T_2 . Merely using the cache life for T_2 list did not fare well, compared to DEMOTE-ARC, in unlimited bandwidth cases. The *probPromote* for the T_2 lists is dynamically adapted at each level so as to equalize *life/occupancy* across the hierarchy (Lines 41-43).

Another hint called the T_2 hint is used along with read requests and replies to indicate that the page should be stored in a T_2 list as it has been seen earlier (Line 46). If any cache decides to keep the page (Line 59), it creates the page in T_2 if T_2 hint is true; else it creates it in T_1 .

E. Handling Multi-path Hierarchies

Since PROMOTE does not significantly alter the local caching policy, extensions to multi-path hierarchies is as simple as requiring that the caches with multiple directly connected higher caches maintain a separate *probPromote* value corresponding to each such higher cache, and that hints be sent to all directly connected lower caches. It may not always be possible to equalize the cache lives or marginal utilities, however, merely allowing the adaptation to attempt the equalization results in better performance.

On the other hand, it is difficult to conceive a multi-path version of DEMOTE in many cases (e.g. ARC hierarchies in Figures 14 and 15). Hence, PROMOTE is not only easier to apply to multi-level caches than DEMOTE, it is also more broadly applicable.

IV. EXPERIMENTAL SET-UP

A. Traces

We use both synthetic and real-life scenario traces that have been widely used for evaluating caching algorithms.

P1-P14: These traces [25], [17] were collected over several months from workstations running Windows NT by using VTrace [23].

Financial1 and Financial2: These traces [16] were collected by monitoring requests to disks of OLTP applications at two large financial institutions.

SPC1: We use a trace (as seen by a subset of disks) when servicing the SPC-1 benchmark [24]. It combines three workloads that follow purely random, sequential, and hierarchical reuse access models. This synthetic workload has been widely used for evaluating cache algorithms [25], [14], [16], [3].

Zipf Like: We use a synthetic trace that follows a Zipf-like [37] distribution, where the probability of the i^{th} page being referenced is proportional to $1/i^\alpha$ ($\alpha = 0.75$, over 400K blocks). This approximates common

access patterns, such as in web traffic [9], [5]. Multi-level caching algorithms [33], [12] have employed this trace for evaluations.

Since write cache management policies need to leverage both temporal and spatial locality (see [15]), the write cache is typically managed using a policy distinct from the read cache. Following previous work [33], [12], we focus on the read component of caches and choose to ignore the writes for simplicity. Including the writes would only turn the comparisons more in favor of PROMOTE as they would increase contention for the disk and network resources, a scenario in which PROMOTE easily outshines DEMOTE. Each trace is limited to the first two million reads to shorten the experiment durations.

B. Software Setup

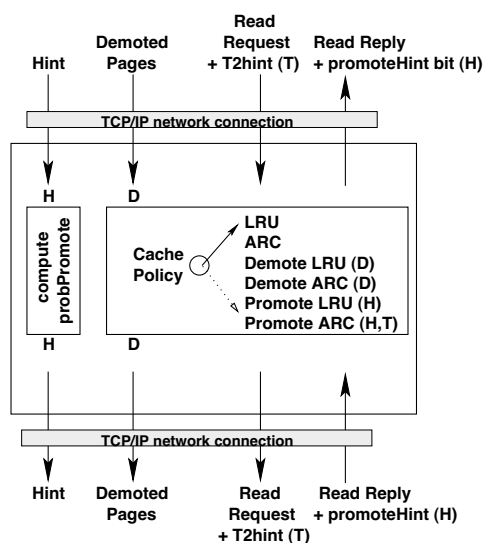


Fig. 7. CacheSim block diagram: Multiple instances form a hierarchy of caches. Algorithm-specific interfaces are marked: D, H, and T (T_2hint).

We implemented *CacheSim* (Figure 7), a framework which allows us to benchmark various algorithms for a wide range of cache sizes, real-life traces, and any single and multi-path hierarchy. *CacheSim* is instantiated with a given cache size and one of the following policies: LRU, ARC, DEMOTE-LRU, DEMOTE-ARC, PROMOTE-LRU, and PROMOTE-ARC. One instance of *CacheSim* simulates one level of cache in a multi-level cache hierarchy, while communicating to the higher and lower caches over a TCP/IP network. The highest level reads requests from a trace file one I/O (size 512 bytes) at a time, with no thinktime, while the lowest level simulates disk responses with a fixed response time.

Apart from the traditional read interfaces, *CacheSim* also implements two special interfaces:

- **DEMOTE interface:** Used only by the DEMOTE technique to transfer evicted pages to the next lower cache.
- **Hint interface:** Used by the PROMOTE technique to periodically transfer life and occupancy information to the next lower cache.

CacheSim simulates the following realistic roundtrip response times for storage system hierarchies (based on our knowledge): For two-level scenarios: $t_1 = 0.5$ ms, $t_2 = 1.0$ ms, $t_m = 5.0$. For three-level scenarios: $t_1 = 0.5$ ms, $t_2 = 1.0$ ms, $t_3 = 2.0$ ms, $t_m = 10.0$ ms. The results in this paper are applicable for any set of values where $t_i < t_{i+1}$.

C. The Competitors: DEMOTE vs. PROMOTE

To compare the two techniques, we apply both the DEMOTE and the PROMOTE techniques to two popular single-level caching policies. While LRU is the most fundamental caching technique (variants of which are widely used [26], [13], [11], [31]), ARC is arguably the most powerful single-level caching algorithm [25]. A multi-level cache performing the same as a single-level ARC cache represents the most powerful application-independent, multi-level caching policy.

DEMOTE-ARC is implemented by maintaining the T_1 and T_2 lists as global lists, divided among all caches in proportion to the size of the caches [12]. The aggregate hit ratio is precisely equal to that obtained by a single cache of the aggregate size implementing ARC. This is the strongest multi-level caching contender we can devise.

DEMOTE-LRU is implemented as suggested in earlier work [33], also depicted earlier in Figure 2. PROMOTE-LRU and PROMOTE-ARC are implemented as explained in Section III.

For completeness, we also compare the performance of LRU, which is defined as the simple Least Recently Used (LRU) policy implemented in each cache within the hierarchy. There are no inclusion or exclusion guarantees since each cache behaving as if it were the only cache.

D. Measuring Success: The Treachery of Hits

Hit ratio has been extensively used in the study of single-level caches, where higher hit ratios generally imply better performance. In multi-level scenarios, however, the benefit of a hit varies significantly depending on the cache level at which it occurs, making hit ratio a misleading performance metric.

In this paper we use the average response time as the key performance metric. In practice, different algorithms result in different amounts of inter-cache traffic, and in limited bandwidth scenarios, the observed average response time depends more on the inter-cache

Two Cache Hierarchy on trace P1

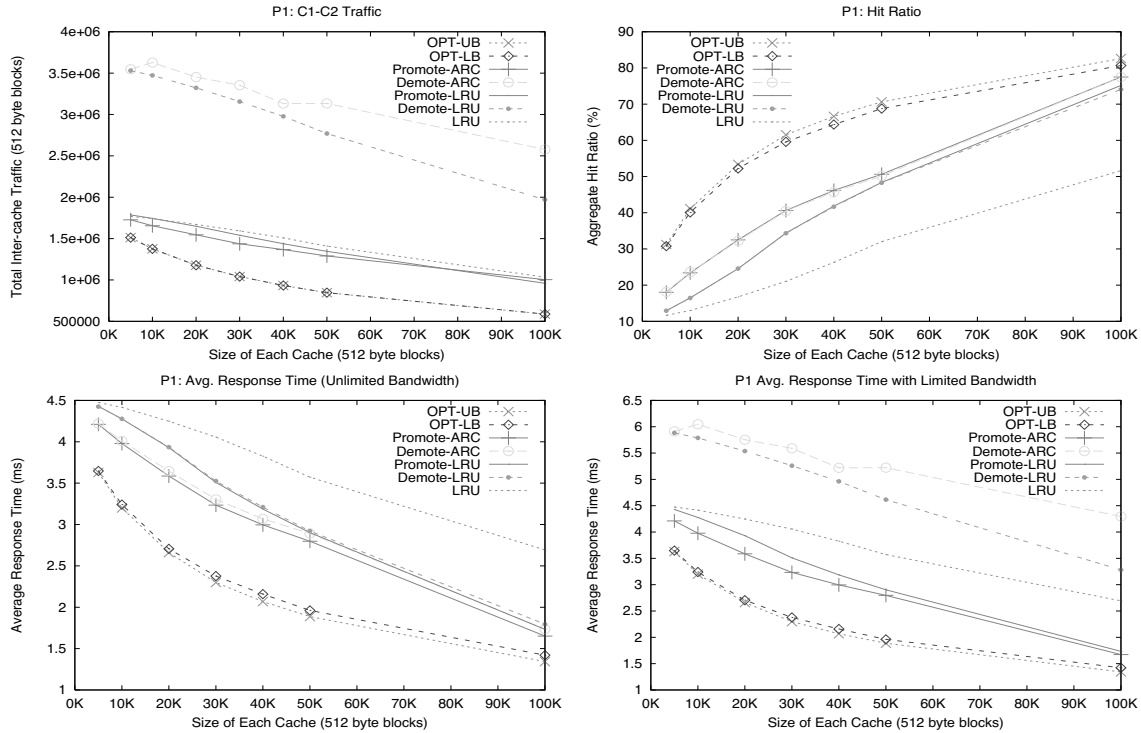


Fig. 8. On the x-axis is the size of the C_1 and C_2 caches. We plot the inter-cache traffic (top-left), the aggregate hit ratio (top-right), and the average response time allowing unlimited bandwidth and free demotions (bottom-left), as a function of the caching algorithm and cache size. In a limited bandwidth scenario (300 blocks per second: 1.5 times that required if there were no hits or demotions), PROMOTE outperforms DEMOTE significantly (bottom-right).

bandwidth usage than on the number or location of hits. Hence, we measure both the inter-cache bandwidth usage and the average response time (assuming unlimited bandwidth). The actual bandwidth limit depends on the hardware and the number and intensity of other applications using the same network fabric. We provide measurements for some limited bandwidth cases as well.

In our experimental results, error bars are not shown since the average response times over separate runs was within 1%, even for the adaptive algorithms.

V. RESULTS

A. Very Close Bounds on Offline Optimal Performance: OPT-UB and OPT-LB

We computed the average response times for OPT-UB and OPT-LB for a wide range of traces (Section IV-A) and cache sizes (Figures 8 through 13), and found that on an average the bounds ran within 2.18% and 2.83% of each other for two and three level single-path hierarchies, respectively. The maximum separation between bounds for any trace and cache combination was only 8.6% and 10.0% for the two and three level caches, respectively. In terms of inter-cache bandwidth usage, OPT-LB is optimal and coincides with OPT-UB

for the C_1 - C_2 traffic. This is because OPT-LB does not use any demotions and achieves the maximum possible hits in the C_1 cache (as given by Belady's OPT). For C_2 - C_3 traffic, the bounds run, on an average, within 3.4% of each other. OPT-LB is not optimal for the C_2 - C_3 traffic because it does not use demotions between the C_1 and C_2 which could have potentially reduced the number of misses flowing out of C_2 .

In a more complex multi-path scenario shown in Figure 14 (and Figure 15), the bounds ran about 8.5% (and 10.8%) of each other in terms of the average response time, and coincided in terms of inter-cache traffic.

We believe that the closeness of these bounds in practice and the fact that they are significantly superior to the current state-of-the-art multi-level caching algorithms (Figures 8 through 13) constitute an extremely significant result, and provide an important missing perspective to the field of multi-level caching.

B. Two Cache Hierarchy

In Figure 8, we present detailed results for a first trace, P1, in a two cache (same size) hierarchy. We observed similar results with all traces given in Section IV-

Two Cache Hierarchy on traces P3, P5, and P7

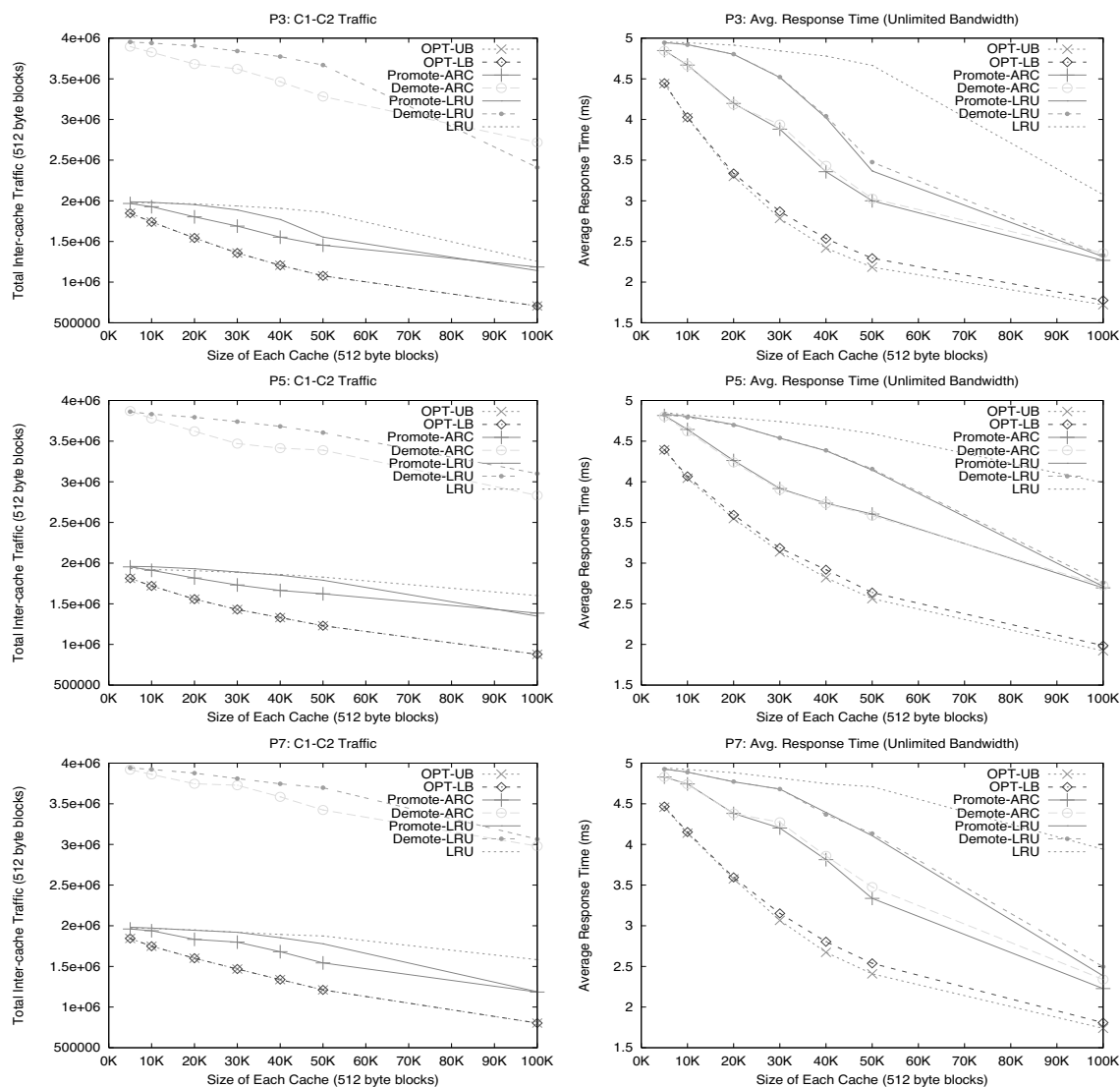


Fig. 9. On the x-axis is the size of the C_1 and C_2 caches. We plot the inter-cache traffic (left), and the average response time allowing unlimited bandwidth and free demotions (right), as a function of the caching algorithm and cache size.

A, the results for some of which are shown in Figures 9 and 10.

Inter-cache Bandwidth Usage: PROMOTE 2x more efficient than DEMOTE. In Figure 8-10, we plot the total traffic between the C_1 and C_2 caches (demotions + C_1 misses). Observe that as the cache sizes grow, the inter-cache traffic decreases as C_1 produces more hits. For both LRU and ARC variants, DEMOTE generates more than double the traffic generated by PROMOTE. This is because DEMOTE causes a demotion for every C_1 miss (after C_1 is full), and also incurs more misses in C_1 than PROMOTE. This is true for all traces and cache sizes, where, on an average, DEMOTE requires 101% more inter-cache bandwidth than PROMOTE for

the LRU variant, and about 121% more for the ARC variant.

Aggregate Hit Ratio: PROMOTE same as DEMOTE. In Figure 8, we observe that both PROMOTE-LRU and PROMOTE-ARC achieve almost the same aggregate hit ratio as their DEMOTE counterparts. This was observed for all traces and cache sizes. We also confirm that plain LRU achieves the lowest aggregate hit ratio as the inclusive nature of the lower cache results in very few hits. Please note, however, that the aggregate hit ratio is not a reliable performance metric.

Hits in the Highest Cache: PROMOTE beats DEMOTE.

For the same aggregate hit ratio, a higher number

Two Cache Hierarchy on traces Financial2, SPC1, and Zipf-like

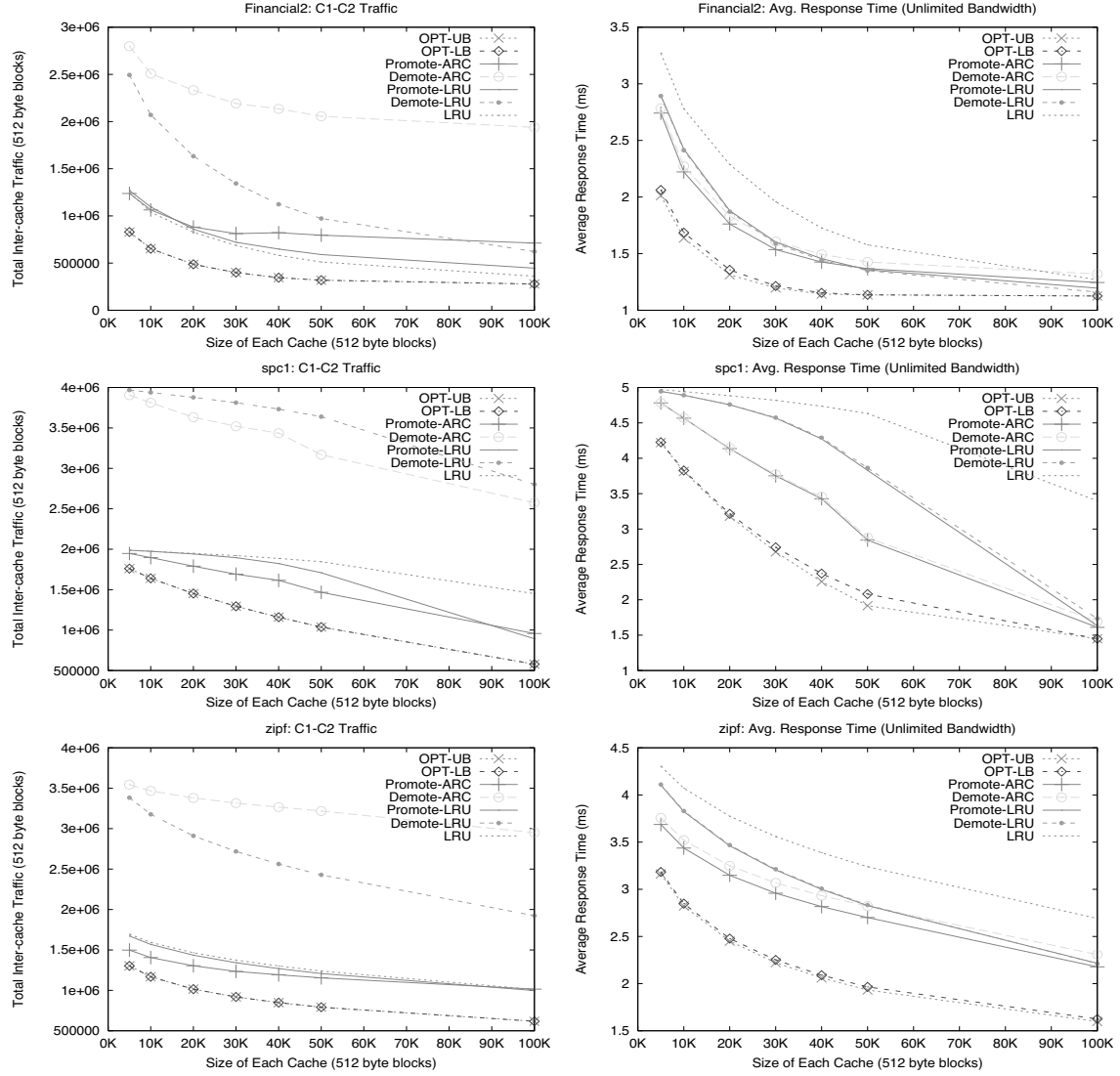


Fig. 10. On the x-axis is the size of the C_1 and C_2 caches. We plot the inter-cache traffic (left), and the average response time allowing unlimited bandwidth and free demotions (right), as a function of the caching algorithm and cache size.

	P1	P3	P5	P7	P9	P11	Financial1	Financial2	spc1	zipf
PROMOTE-ARC	709476	546440	377332	456983	570250	692042	532270	1204243	533414	845104
DEMOT-ARC	408174	333751	280481	263244	469771	540978	506503	947009	390810	365909
PROMOTE-LRU	653880	446803	210473	222156	511271	639430	644893	1410101	291183	791667
DEMOT-LRU	590276	140384	172625	125960	396135	667633	688946	1488752	155197	760877

TABLE I. Number of C_1 hits (out of 2000000 requests), at a cache size of 50K blocks. While aggregate hits were almost the same for both PROMOTE and DEMOTE, we observed that PROMOTE accumulates more hits in C_1 .

Varying Relative Cache Sizes in a Two Cache Hierarchy

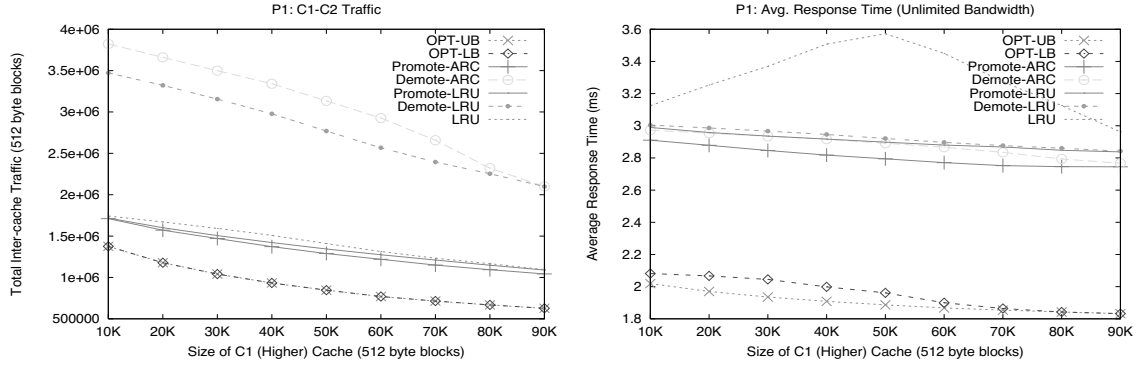


Fig. 11. On the x-axis is the size of the C_1 (higher) cache. $|C_1| + |C_2| = 100K$ blocks.

of hits in the highest cache leads to better average response times. In Table I, we compare the number of hits in C_1 for a wide range of traces with two levels of cache of 50K blocks each. While all the traces exhibited similar behavior, we skip some traces in the table to keep it small. We observe that PROMOTE-LRU beats DEMOTE-LRU by 13.0% on an average, and PROMOTE-ARC beats its DEMOTE contender by 37.5%. This is primarily because PROMOTE probabilistically promotes high reuse pages to the higher cache, while DEMOTE forces all pages to flow through the higher cache, pushing out a great number of hits to the lower cache levels.

Average Response Time: PROMOTE beats DEMOTE.

In Figure 8 we plot the average response time for the trace P1 at various cache sizes. When we assume an unlimited inter-cache bandwidth and free demotions (although DEMOTE has a rough model to attribute for demotion costs, we create the case most favorable to DEMOTE), PROMOTE-LRU still beats DEMOTE-LRU by up to 4% and PROMOTE-ARC beats DEMOTE-ARC by up to 5% across all cache sizes. For all other traces (some shown in Figures 9 and 10) PROMOTE achieves 0.3%(LRU) and 1.5%(ARC) better response times on an average.

In the lower right panel of Figure 8, we examine a limited bandwidth case, which is more realistic. We allow 300 blocks per second, which is 1.5x that required if there were no hits or demotions ($1/t_m = 200$). When we average the response time across all cache sizes, we observe that PROMOTE substantially outperforms DEMOTE by achieving lower response times, 3.21ms (for ARC) and 3.42ms (for LRU), as compared to DEMOTE with 5.43ms (for ARC) and 5.05ms (for LRU), respectively. In fact, both DEMOTE-LRU and DEMOTE-ARC consistently perform worse than even plain LRU. Surprisingly, for smaller cache sizes, DEMOTE variants perform even worse than no caching at

all (i.e. worse than $t_m = 5ms$)! This happens because, when bandwidth is the bottleneck and we use DEMOTE, the bandwidth saved due to one hit in cache C_1 is consumed by one demotion due to a miss. When the number of misses is greater than the number of hits in the cache C_1 ($< 50\%$ hit ratio), a no-caching policy will actually perform better.

Since DEMOTE is clearly worse in the limited bandwidth case, we consistently assume unlimited inter-cache bandwidth and free demotions for the remaining traces shown in Figures 9 and 10.

C. Differing Cache Sizes

In Figure 11, we vary the relative size of the C_1 and C_2 caches from 1 : 9 to 9 : 1 while keeping the aggregate cache size equal to 100K blocks. We present average response time and inter-cache traffic (assuming unlimited bandwidth and free demotions) for the trace P1 (other traces have similar results). We observe that PROMOTE variants have consistently better response times than the DEMOTE variants across the entire spectrum of relative sizes. The average response time for plain LRU peaks (implying that the hit ratio is the lowest) when the two caches are of the same size. This confirms that the most duplication of data happens when the caches are of comparable sizes. For all the other algorithms, the average response time decreases as the size of the C_1 cache increases as more hits occur at the highest cache.

As before, we observe that the DEMOTE variants invariably consume 2x bandwidth when compared to the PROMOTE variants.

D. Varying Inter-cache Bandwidth

We consider a client using 50K blocks each in two levels of cache, and having a network impact of up to 500KBps. In a typical enterprise SAN environment, there are thousands of such concurrent application threads, scaling the need for both cache and

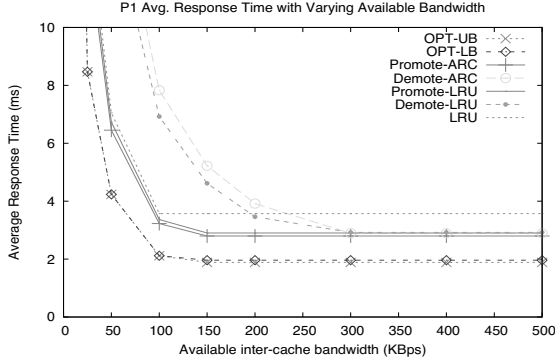


Fig. 12. In a two-cache hierarchy with 50K blocks each, we vary the inter-cache bandwidth available to a client on the x-axis. On the y-axis we plot the average response time in milliseconds.

network resources by many orders of magnitude. In fact, even without demotions, many SAN environments are regularly found bandwidth-bound (particularly with sequential reads), implying, as we observe below, that implementing DEMOTE might be detrimental to performance.

In Figure 12, we plot the average response time as a function of the inter-cache bandwidth available to the client. For each algorithm, we notice two regimes, a bandwidth-sensitive regime on the left side where decreasing the bandwidth available increases the average response time, and a bandwidth-insensitive, flat regime, on the right. As expected, OPT-UB, closely followed by OPT-LB, performs the best, with the lowest average response time and the least inter-cache bandwidth requirement (as indicated by the long flat portion on the right). Note that, the DEMOTE variants perform even worse than plain LRU when bandwidth is not abundant. SAN environments which cannot accommodate the 2x network cost of DEMOTE over LRU will see this behavior. This fundamental concern has limited the deployment of DEMOTE algorithms in commercial systems.

The PROMOTE variants are significantly better than the DEMOTE variants when bandwidth is limited, while they outperform LRU handsomely when bandwidth is abundant. As bandwidth is reduced, LRU becomes only marginally worse than PROMOTE because the benefit of more hits in the lower cache, C_2 , is no longer felt as the bandwidth between the caches becomes the bottleneck. Overall, PROMOTE performs the best in all bandwidth regimes.

E. Three Cache Hierarchy

Increasing the complexity of the hierarchies we study, we now turn to a three-level (three equal size caches) hierarchy.

As in the two-level case, we present detailed results for the first trace P1 in Figure 13. The other traces had similar results but we do not present plots for lack of space.

We observe that for the wide variety of traces and cache sizes, PROMOTE outperforms DEMOTE in three-level caches as well:

Inter-cache Bandwidth Usage: PROMOTE is 2x more efficient than DEMOTE which uses 105% (111% resp.) more bandwidth between C_1 and C_2 and 98% (113% resp.) more bandwidth between C_2 and C_3 , when compared to PROMOTE-LRU (PROMOTE-ARC, respectively).

Aggregate Hit Ratio: PROMOTE same as DEMOTE.

Hits in the Highest Cache: PROMOTE achieves 1.5% and 10% more hits in the top two caches than DEMOTE for the LRU and ARC variants, respectively.

Average Response Time: When bandwidth is not limited and demotions are free, PROMOTE beats DEMOTE by 0.2% and 1.3% on the average response time for LRU and ARC variants, respectively. For a limited bandwidth case, where we allow 200 blocks per second ($2x$ times $1/t_m = 100$), When we average the response time across all cache sizes, we observe that PROMOTE substantially outperforms DEMOTE by achieving lower response times, 5.61ms (for ARC) and 5.93ms (for LRU), as compared to DEMOTE with 8.04ms (for ARC) and 7.57ms (for LRU), respectively.

F. More Complex Cache Hierarchies

PROMOTE can be applied to complex hierarchies. As the possible configurations are endless, we pick two simple and yet interesting configurations for our experiments.

1) Tree-like Hierarchy: We use a hierarchy of three caches, C_{1a} (40K blocks) and C_{1b} (30K blocks) at the first level, and a shared cache C_2 (50K blocks) at the second-level (see Figure 14). While C_{1a} serves one of P2, P3, P4 or P5 traces, C_{1b} serves the P1 trace. We impose no bandwidth restrictions and assume free demotions for the DEMOTE algorithm. We observe that for all four combinations, the PROMOTE-LRU has equal or better average response time while generating only half the inter-cache traffic than DEMOTE-LRU. DEMOTE cannot be applied to tree-like ARC hierarchies, allowing PROMOTE-ARC to win by default. This is because DEMOTE simulates a global ARC algorithm which adapts the ratio of the global ARC lists, T_1 and T_2 . In single-path scenarios, the same ratio of the two ARC lists, $|T_1| : |T_2|$, can be enforced at all levels. However, in multi-path scenarios, this is not always possible, as the amount of T_2 pages in a cache depends on the workload it sees, and the ratio determined by ARC may not be enforceable.

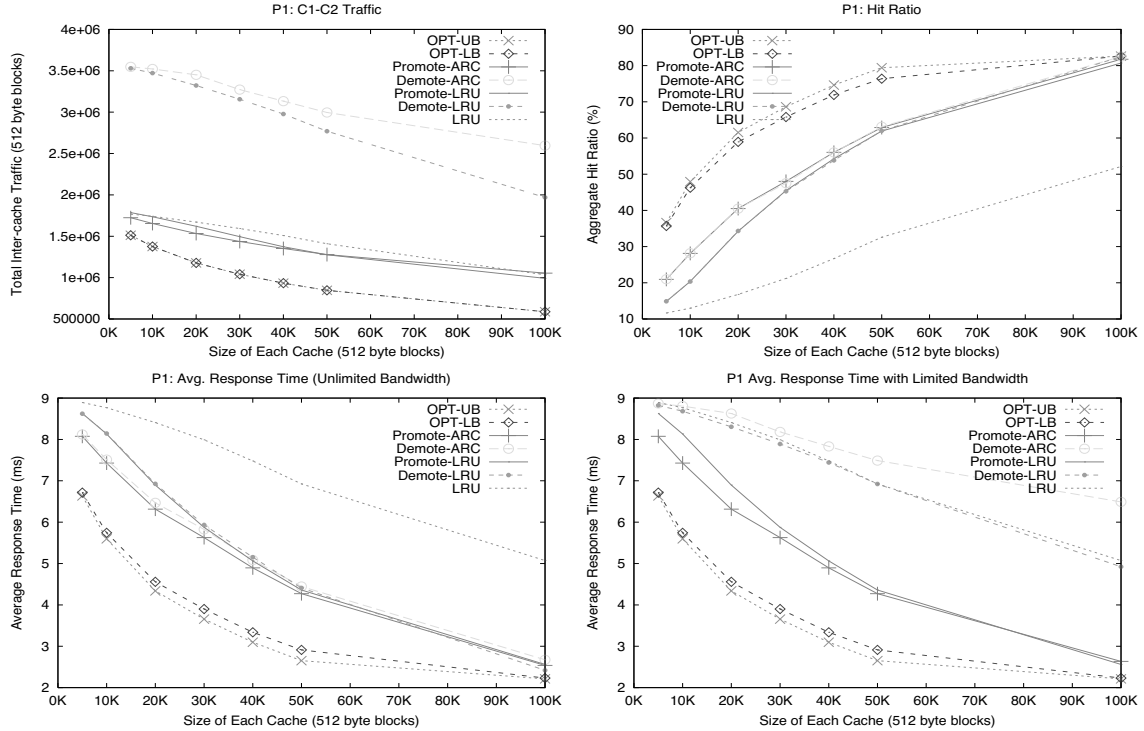
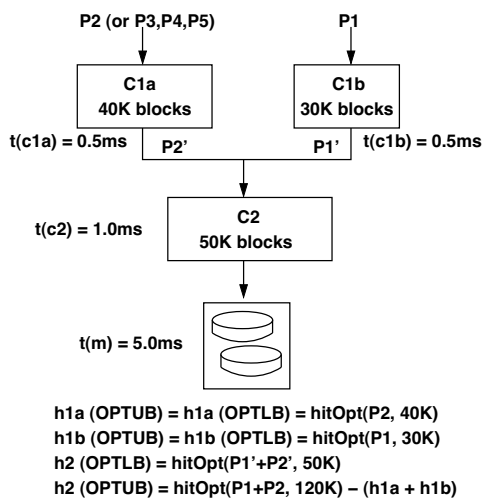
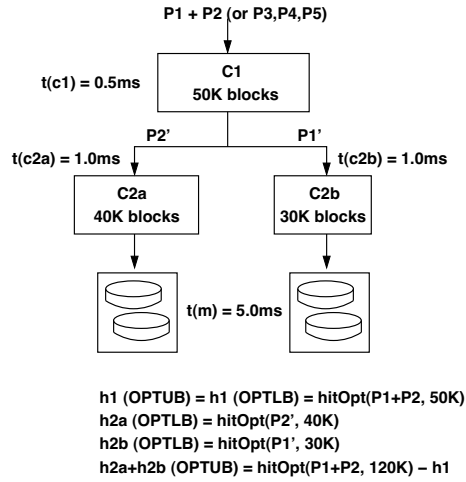


Fig. 13. On the x-axis is the size of each cache: C_1 , C_2 , and C_3 . We present the total traffic between C_1 and C_2 (top-left), the aggregate hit ratio (top-right), and the average response time allowing unlimited bandwidth and free demotions (bottom-left) for a range of per-level cache sizes. In a limited bandwidth scenario (200 blocks per second: 2 times that required if there were no hits or demotions), PROMOTE outperforms DEMOTE significantly (bottom-right).



		Hit Ratio Contribution = hits / 4000000				Traf. (MBlocks)		Avg. Resp. Time
		C1a	C1b	C2	Aggr	C1a- C2	C1b- C2	
P1+P2	DEMOT-LRU	0.14	0.10	0.12	0.36	2.83	3.16	3.45
	PROMOT-LRU	0.14	0.12	0.10	0.36	1.46	1.54	3.45
	PROMOT-ARC	0.16	0.14	0.13	0.43	1.36	1.43	3.13
	OPT-LB	0.27	0.24	0.10	0.61	0.90	1.04	2.29
	OPT-UB	0.27	0.24	0.12	0.63	0.90	1.04	2.12
P1+P3	DEMOT-LRU	0.02	0.10	0.10	0.23	3.77	3.16	4.01
	PROMOT-LRU	0.04	0.11	0.08	0.23	1.83	1.55	4.01
	PROMOT-ARC	0.10	0.14	0.10	0.34	1.59	1.45	3.51
	OPT-LB	0.20	0.24	0.12	0.56	1.21	1.04	2.53
	OPT-UB	0.20	0.24	0.14	0.59	1.21	1.04	2.29
P1+P4	DEMOT-LRU	0.02	0.10	0.06	0.18	3.82	3.16	4.23
	PROMOT-LRU	0.01	0.11	0.06	0.18	1.94	1.55	4.22
	PROMOT-ARC	0.03	0.14	0.10	0.27	1.87	1.45	3.85
	OPT-LB	0.09	0.24	0.08	0.41	1.66	1.04	3.20
	OPT-UB	0.09	0.24	0.11	0.44	1.66	1.04	2.98
P1+P5	DEMOT-LRU	0.03	0.10	0.08	0.21	3.68	3.16	4.08
	PROMOT-LRU	0.04	0.13	0.05	0.22	1.86	1.55	4.06
	PROMOT-ARC	0.09	0.14	0.09	0.32	1.66	1.43	3.60
	OPT-LB	0.17	0.24	0.10	0.51	1.33	1.04	2.74
	OPT-UB	0.17	0.24	0.12	0.53	1.33	1.04	2.53

Fig. 14. A tree-like multi-path cache hierarchy. Traces do not overlap (2 million reads each). For ease of comparison between caches, the individual hit ratio contributions are normalized based on the total number of reads in the cache hierarchy.



		Hit Ratio Contribution = hits / 4000000				Traf. (MBlocks)		Avg. Resp. Time
		C1	C2a	C2b	Aggr	C1- C2a	C1- C2b	
P1+P2	DEMOTÉ-LRU	0.19	0.10	0.06	0.35	3.17	3.27	3.50
	PROMOTE-LRU	0.18	0.11	0.06	0.35	1.73	1.54	3.49
	PROMOTE-ARC	0.23	0.13	0.07	0.43	1.62	1.46	3.18
	OPT-LB	0.46	0.06	0.08	0.60	1.05	1.09	2.35
	OPT-UB	0.46	0.00	0.00	0.63	1.05	1.09	2.10
P1+P3	DEMOTÉ-LRU	0.11	0.08	0.07	0.25	3.83	3.25	3.93
	PROMOTE-LRU	0.15	0.04	0.05	0.24	1.87	1.55	3.97
	PROMOTE-ARC	0.20	0.08	0.06	0.35	1.78	1.40	3.52
	OPT-LB	0.40	0.06	0.10	0.56	1.33	1.08	2.57
	OPT-UB	0.40	0.00	0.00	0.59	1.33	1.08	2.27
P1+P4	DEMOTÉ-LRU	0.13	0.01	0.06	0.20	3.87	3.04	4.15
	PROMOTE-LRU	0.11	0.02	0.06	0.19	1.96	1.58	4.20
	PROMOTE-ARC	0.17	0.03	0.06	0.26	1.96	1.37	3.88
	OPT-LB	0.32	0.05	0.03	0.41	1.75	0.95	3.21
	OPT-UB	0.32	0.00	0.00	0.44	1.75	0.95	2.98
P1+P5	DEMOTÉ-LRU	0.12	0.04	0.07	0.22	3.75	3.27	4.07
	PROMOTE-LRU	0.13	0.03	0.06	0.22	1.91	1.58	4.07
	PROMOTE-ARC	0.18	0.07	0.07	0.31	1.86	1.44	3.66
	OPT-LB	0.36	0.06	0.09	0.51	1.49	1.06	2.77
	OPT-UB	0.36	0.00	0.00	0.53	1.49	1.06	2.51

Fig. 15. An inverted tree-like single-path cache hierarchy. Two traces are merged before presenting to cache C_1 . For ease of comparison between caches, the individual hit ratio contributions are normalized based on the total number of reads in the cache hierarchy.

In Figure 14, we also show the steps used to compute OPT-UB and OPT-LB in accordance to Section II-D. As usual, we observe that OPT-LB and OPT-UB provide close bounds (8.5% apart) on the optimal performance for the given hierarchy.

2) *Inverted Tree-like Hierarchy*: We invert the cache hierarchy used above as shown in Figure 15. At the first level we have a single cache C_{1a} (50K blocks) and at the second level we have C_{2a} (40K blocks) and C_{2b} (30K blocks). The trace P1 accesses data through the C_1, C_{2a} hierarchy, while the traces P2, P3, P4 or P5 are served through the C_1, C_{2b} hierarchy. We again notice that PROMOTE-LRU performs within 1% of the DEMOTÉ variant in terms of response time, and is twice as efficient in terms of bandwidth usage. PROMOTE-ARC performs much better than the LRU based algorithms as expected. DEMOTÉ cannot generalize ARC for this hierarchy and thus is not a contender. Again we observe that OPT-LB and OPT-UB provide close bounds (10.8% apart) on the optimal performance for the given hierarchy.

VI. CONCLUSIONS

As large caches are becoming ubiquitous, multi-level caching is emerging as an important field for innovation. In this paper we have made two major contributions.

We have demonstrated a simple and powerful technique, called PROMOTE, which is significantly superior to the popular DEMOTÉ technique. For half the bandwidth, PROMOTE provides similar aggregate hit ratios for a variety of workloads, with more hits in the topmost cache. The reduction in bandwidth usage provides huge improvements in average response times when network resources are not abundant. Even in

constrained bandwidth cases, unlike DEMOTÉ, PROMOTE always performs better than a non-exclusive hierarchy of caches. This characteristic is essential for implementation in commercial systems where network usage behavior cannot be predicted. We anticipate the principles in the PROMOTE technique to engender more sophisticated multi-level caching policies in the future.

While improving caching algorithms is important, knowing the theoretical bounds on performance is extremely invaluable. We have provided this much needed knowledge in the form of two very close bounds on the optimal performance. OPT-UB delimits the best possible response time and bandwidth usage for any multi-level caching policy, while, OPT-LB serves as the best known off-line multi-level caching policy that we are aware of. We hope that these new bounds will spur and guide future research in this field.

VII. ACKNOWLEDGMENTS

We are indebted to the anonymous reviewers of the paper for their insightful comments. We are also grateful to Dr. Theodore Wong, our shepherd, for detailed comments and suggestions that greatly improved the readability of the paper.

REFERENCES

- [1] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
- [2] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

- [3] S. Bansal and D. Modha. Car: Clock with adaptive replacement, 2004.
- [4] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Sys. J.*, 5(2):78–101, 1966.
- [5] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.
- [6] Z. Chen, Y. Zhou, and K. Li. Eviction-based cache placement for storage caches. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 269–282, 2003.
- [7] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 145–156, New York, NY, USA, 2005. ACM Press.
- [8] F. J. Corbató. A paging experiment with the multics system. In *In Honor of P. M. Morse*, pages 217–228. MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.
- [9] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE /ACM Transactions on Networking*, 5(6):835–846, 1997.
- [10] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.
- [11] EMC. EMC symmetric dmx architecture guide. http://www.emc.com/products/systems/pdf/C1011_emc-symm-dmx-pdg-ldv.pdf, March 2004.
- [12] Michael Factor, Assaf Schuster, and Gala Yadgar. Karma: Know-it-all replacement for a multilevel cache. In *Proc. of the USENIX Conference on File and Storage Technologies*, 2007, 2007.
- [13] Kevin W. Froese and Richard B. Bunt. The effect of client caching on file server workloads. In *HICSS (1)*, pages 150–159, 1996.
- [14] Binny S. Gill and Dharmendra S. Modha. SARC: Sequential prefetching in adaptive replacement cache. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 293–308, 2005.
- [15] Binny S. Gill and Dharmendra S. Modha. WOW: Wide ordering of writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 129–142, 2005.
- [16] Pawan Goyal, Divyesh Jadav, Dharmendra S. Modha, and Renu Tewari. CacheCOW: providing QoS for storage system caches. In *SIGMETRICS*, pages 306–307, 2003.
- [17] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. *ACM Trans. Comput. Syst.*, 23(4):424–473, 2005.
- [18] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS Conf.*, 2002.
- [19] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. *ICDCS*, 00:168–177, 2004.
- [20] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. VLDB Conf.*, pages 297–306, 1994.
- [21] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers*, 50(12):1352–1360, 2001.
- [22] Lishing Liu. Issues in multi-level cache designs. In *ICCS '94: Proceedings of the 1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, pages 46–52, Washington, DC, USA, 1994. IEEE Computer Society.
- [23] J. R. Lorch and A. J. Smith. The VTrace tool: Building a system traces for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, Oct 2000.
- [24] Brue McNutt and Steven Johnson. A standard test of I/O cache. In *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.
- [25] Nimrod Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, pages 115–130, 2003.
- [26] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.
- [27] Li Ou, Xubin He, Martha J. Kosa, and Stephen L. Scott. A unified multiple-level cache for high performance storage systems. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.
- [28] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance-optimal multi-level cache hierarchies. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 114–121, New York, NY, USA, 1989. ACM Press.
- [29] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS Conf.*, pages 134–142, 1990.
- [30] R. T. Short and H. M. Levy. A simulation study of two-level caches. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 81–88, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
- [31] J. Z. Teng and R. A. Gumaer. Managing IBM database 2 buffers the effect of client caching on file server workloads. *IBM Systems Journal*, 23(2):211–218, 1984.
- [32] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt. Disk cache replacement policies for network filesystems. In *International Conference on Distributed Computing Systems*, pages 2–11, 1993.
- [33] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proc. of the USENIX Annual Technical Conference*, pages 161–175, 2002.
- [34] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Demotion-based exclusive caching through demote buffering: Design and evaluations over different networks. In *Workshop on Storage Network Architecture and Parallel I/O (SNAPI)*, 2003.
- [35] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):505–519, 2004.
- [36] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, 2001.
- [37] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison Wesley, Reading, MA, 1949.

AWOL: An Adaptive Write Optimizations Layer

Alexandros Batsakis, Randal Burns

The Johns Hopkins University

Arkady Kanevsky, James Lentini, Thomas Talpey

Network ApplianceTMInc.

Abstract

Operating system memory managers fail to consider the population of read versus write pages in the buffer pool or outstanding I/O requests when writing dirty pages to disk or network file systems. This leads to bursty I/O patterns, which stall processes reading data and reduce the efficiency of storage. We address these limitations by adaptively allocating memory between write buffering and read caching and by writing dirty pages to disk opportunistically before the operating system submits them for write-back. We implement and evaluate our methods within the Linux® operating system and show performance gains of more than 30% for mixed read/write workloads.

1 Introduction

Scaling trends in processor speed and disk performance (access time and throughput) have brought write performance into the critical path. Traditionally, reads have been considered more important than writes: appropriately given that reads are synchronous and writes are generally asynchronous. We refer to I/Os as synchronous and asynchronous to describe whether the issuing applications or the operating system is blocking awaiting their completion. Most cache-management algorithms focus on directing the population of read pages [11, 12, 17, 20]. However, as processors increase in speed, systems have the ability to create dirty pages at rates well beyond the disk's ability to clean them. Gill et al. [8] point out an annual growth rate of 60% for processors and 8% for disk access time. In such an environment, the synchronous reads depend upon the asynchronous writes, because (1) dirty pages consume memory that is unavailable for read caching

and (2) write traffic to clean pages interferes with read requests. In a sense, there is a priority inversion [23] between reads and writes to which we need to apply priority scheduling, preferring reads to writes, and priority inheritance, performing writes that block high priority reads.

The static write and flush policies used by operating system memory managers are insensitive to processes that are actively reading data, the distribution of read versus write pages in the buffer pool, and outstanding I/O requests. This leads to bursty I/O patterns, which both stall other processes reading data and reduce the efficiency of storage. For different workloads, the operating system destages pages either too aggressively or not aggressively enough. We give several examples in Section 2.

Operating system caching is no longer a read-only problem. Operating system memory managers need to be enhanced to balance read and write workloads and to define adaptive destaging policies that are sensitive to workload and the population of read, write and free pages in memory. Recent research has identified the importance of improving write performance. Most of this work addresses write performance independent of reads, e.g. through improved scheduling [8] or separate non-volatile caches [4, 6, 22]. Works that consider reads and writes combined do so in the context of second-tier caches in order to determine what written data are likely to be read again [14].

We define an adaptive framework for destaging dirty pages to disk that reduces the interference of write traffic on read performance and increases the performance of the I/O subsystem. Depending on the workload, it controls the aggressiveness of the destaging policy in order to keep memory available in the page cache and increases disk throughput, while having a minimal impact on cache hit rates. The framework dynamically tunes the allocation of memory between read and write pages. To do so, it employs several techniques:

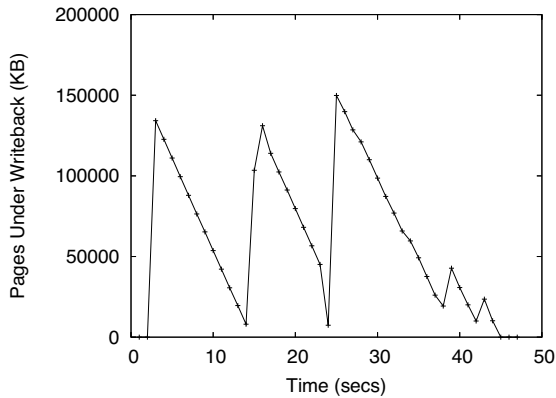


Figure 1: Number of pages under write-back when writing a 512MB file

- (1) **smart scheduling of asynchronous writes** that adapts operating system destaging policies based on available memory and workload;
- (2) **balancing the population of read and write pages** through multiple ghost caches that track the allocation of memory; and,
- (3) **opportunistic write scheduling** using a unified memory/device scheduler I/O queue that allows the I/O system to actively flush dirty pages prior to the pages being submitted for write-back.

The framework does not define new algorithms for managing a read cache. Rather, it works with existing algorithms, e.g. ARC [17], LIRS [11], 2Q [12], adjusting the memory available for read caching. Also, these techniques do not affect the reliability or durability of data in that all operating systems policies, e.g. periodic update deadlines, are enforced.

To demonstrate the benefits of these techniques, we perform experiments based on microbenchmarks and macrobenchmarks that capture a wide range of workloads. Depending on the workload we are able to improve system throughput by more than 30% on average. Finally, the results show that our optimizations are valid not only for local file systems but also for network file systems, such as NFS.

2 Background: Deferred Writing

We give several examples in which the static policies used in operating systems result in poor system performance. We do so by demonstrating that different workloads require different parameterization of the same system variables. Our treatment focuses on Linux, but applies to other operating systems as well. It is also valid for both local and network file systems.

Modern operating systems defer the writing of dirty memory buffers to storage because this noticeably improves performance. In doing so, multiple writes to the

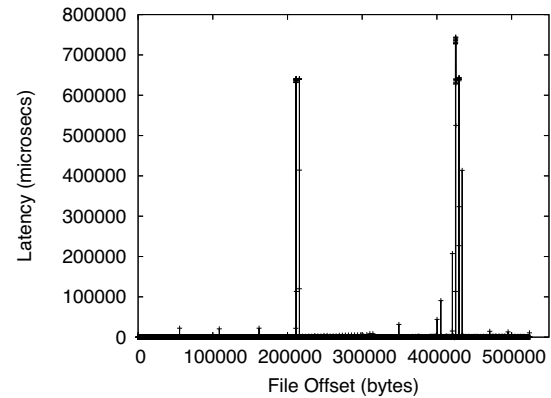


Figure 2: Write latency at each file offset when writing a 512MB file

same page may be aggregated and performed in a single update. Also, spatially related writes may be performed together, even when they occur at different times. Write operations are less critical than read operations, because a process is not suspended as a consequence of a write system call, whereas delayed reads block processes.

A dirty page might stay in memory until the last possible moment—sometimes system shutdown. However, keeping the page in volatile memory for a long period of time has two major drawbacks: first, in case of a failure all unstable updates will be lost and, second, dirty writes occupy memory pages that could be better used for read caching or other purposes, such as anonymous paging.

Traditional UNIX®[25] systems use a periodic update policy in which individual dirty blocks are flushed when their age reaches a predefined limit [18]. Modern systems use an additional criterion to decide when to destage dirty pages to storage. When the number of dirty pages in memory exceeds a certain percentage—the system-wide parameter *dirty.background_ratio* in Linux—a flushing daemon wakes up and starts writing dirty pages to disk until an adequate number of dirty pages have reached storage. By this operation, the flushing daemon ensures that there are always enough free pages available in order to allocate more memory to satisfy new reads and writes.

If applications dirty pages faster than the daemon flushes them to storage, the system will eventually reach the *memory pressure* state: the point at which the maximum allowed number of dirty pages in the system has been reached. Typically this limit is set close to half the size of the available RAM. Memory pressure hampers performance severely because it blocks all writing applications until there is free space for dirty buffers in RAM, which effectively makes all write operations synchronous. Memory pressure has an effect on reads as well. All pending writes that must be

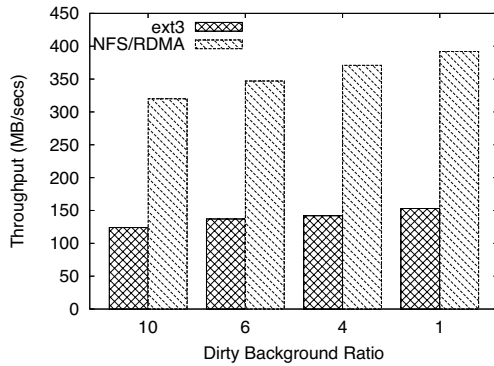


Figure 3: Throughput when writing a 2GB file sequentially

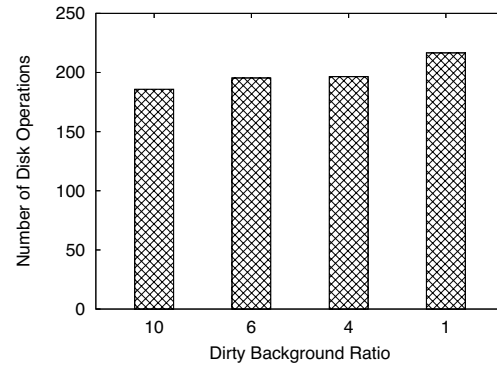


Figure 5: Average number of disk operations per second when compiling Apache

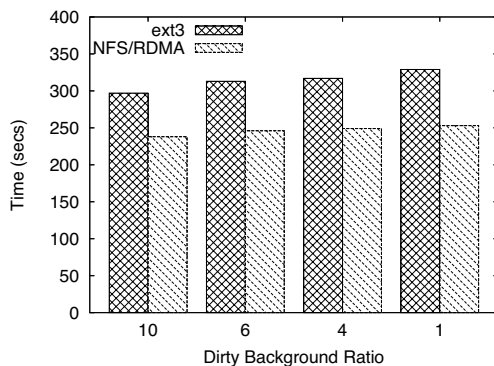


Figure 4: Time to compile Apache for ext3 and NFS over RDMA

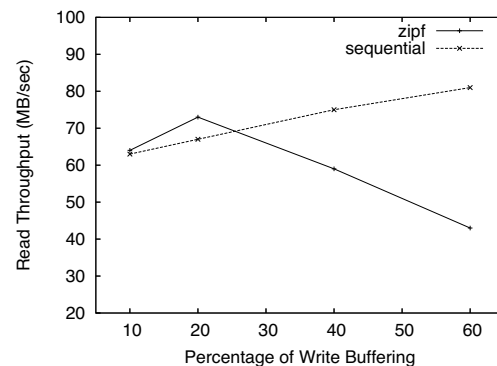


Figure 6: Read throughput for Zipf and sequential access patterns

written to storage interfere with concurrent reads, which results in queuing delays at the device level.

Figure 1 highlights the behavior of asynchronous writes. It shows the number of pages under write-back as a function of time. During a large file write, dirty pages are accumulated in memory until the number of dirty pages exceeds the *dirty_background_ratio* threshold. Then (after a few seconds in Figure 1), the flushing thread wakes up and starts writing pages to disk until their number is below the threshold. The remaining pages are written to the device when the period that dirty pages are allowed to stay in memory expires. Figure 2 shows that memory pressure occurs at two points in time while writing the file, producing significant increases in latency and, thus, a drop in the overall throughput.

The value of the *dirty_background_ratio* that triggers the flushing daemon to start the write-out is critical to system performance. Figure 3 shows the throughput of a process for writing a 2GB file sequentially under different values of the flushing threshold. The highest throughput occurs when the background updates start as soon as possible. The early start of the flushing operation ensures that there are always enough available

memory pages. Figure 4 shows the time required to unzip and compile the Apache web server under various values of the same parameter. A less aggressive flushing policy yields the best performance. This is because the specific workload exhibits many short-lived files and block overwrites. More importantly, keeping the data in memory longer lowers the number of disk I/Os, a practice that makes buffering essential in the presence of concurrent I/O intensive processes (Figure 5).

The effect of the parameters on system throughput is also apparent in the case of NFS/RDMA over Infini-band. The high bandwidth of the device, significantly higher than the disk bandwidth especially in the case of an asynchronous NFS server, makes write buffering undesirable (Figures 3,4). Thus, the capability of the storage devices should be another factor in deciding when to start the write-out process.

An important, but over-looked, issue is the interference of writes and reads at the memory level. Asynchronous writes create dirty buffers that stay in memory for a period of time. This effectively reduces the number of blocks cached as a result of previous read operations. Most systems deal with the sharing of RAM

between write (dirty) and read (cached) pages by setting a hard limit on the maximum number of dirty blocks in memory in order to preserve space for frequently or recently accessed clean pages used to satisfy reads. This hard limit defines the memory pressure point. Thus, the decision of how much memory to use for buffering writes is again critical to the system throughput. Using more memory for reads allows for more cache. On the other hand, it causes the system to reach memory pressure faster.

Figure 6 demonstrates the sensitivity of throughput to the amount of write buffering allowed by the system. These IOzone [10] microbenchmarks show the throughput of a process that performs read and write I/O to two separate 512MB files. In one case, reads are sequential. In the other, the location of reads is drawn from a Zipf distribution. In both cases, writes are sequential. For sequential reads, more write buffering results in less competition for the disk bandwidth between read and background write requests, and, thus, improves performance. For Zipf reads, a very aggressive write buffering policy reduces the cache hit rate of read requests and hampers performance. However, too little write buffering does not yield optimal throughput either, because of the asynchronous nature, writes represent a background load on disk. The obtrusiveness of this load is a significant factor on the experienced throughput of all concurrent synchronous requests. The effect of the write traffic on the system performance is important even though, from the application perspective, writes are completed as soon as the kernel marks the modified buffers dirty.

Another performance concern—unrelated to system parameters and valid for local file systems only—is the selection of dirty pages to destage. The memory manager does not consider the I/O cost when selecting the dirty pages to write back to the device. Instead, it selects pages in the order they were accessed and submits them to the I/O scheduler which chooses the sequence of requests to be sent to disk. The memory manager is oblivious to the I/O cost because it has no information about which pages reside in the scheduler queue. On the other hand, the scheduler is able to reorganize only the buffers that reside on its queues; it has no knowledge of the memory contents. Dirty buffers remain in memory until the flushing policy selects the corresponding page to be written out to the device. During this period, the I/O scheduler may serve I/O requests that are physically close to the block locations of dirty buffered pages. Dirty memory pages, although proximal to the blocks that are about to be dispatched to the storage device, cannot be scheduled for writing. This behavior results in extra disk traffic and increases the number of disk seeks, because the

dirty pages will eventually reach the disk at a later time when their flushing condition is met.

Our work modifies the destaging mechanisms of modern operating systems in order to improve the performance of asynchronous writes and, at the same time, to ameliorate the effects of the competition between synchronous and asynchronous requests at the device and at the memory level. We adaptively tune the write-back process to take into account current workload patterns and, as a result, improve write bandwidth and latency. Also, we develop an algorithm that shares memory between read and write pages, preserving the performance of read caching while improving write throughput. Finally, we implement an architectural change that integrates the memory manager and the I/O scheduler in order to make asynchronous writes less obtrusive, thus, reducing the interference with reads at the disk level.

3 Adaptive Write Scheduling

We propose a new, adaptive, destaging algorithm for volatile caches that manage pages in a read-write cache. Previous studies propose a variety of adaptive algorithms for write-only, non-volatile caches all of which attempt to maintain the cache occupancy as high as possible in order to optimize the order of writes to disk and minimize the I/O operations [3, 19, 27]. In non-volatile caches, pages are stable as soon as they reach memory. Thus, no deadline is assigned to a buffer and there are no starvation nor reliability issues. These methods are not directly applicable to our memory model.

3.1 A Write-Only Cache

At first, we will focus on the simple case where the memory is used only to accommodate writes. Subsequently, we enhance the algorithm to optimize performance in a unified read-write cache.

A destaging algorithm for a write-only memory cache should keep the cache occupancy as high as possible to enable ordering optimizations when dispatching pages. At the same time, it should avoid cache overflow so that no synchronous writes are forced (memory pressure). For a destaging mechanism to be effective, write requests should either be a cache hit or empty space should be available in the cache so that a new page allocation succeeds.

Previous work has compared a number of different destaging algorithms and has shown the performance benefits of a high-low watermark algorithm [27]. When the high threshold is crossed, the memory manager starts writing data out to disk until the percentage of dirty blocks is below the low threshold. The combination of two thresholds controls the start and stop of the

flushing of dirty buffered pages.

The most significant drawback of the high-low watermark scheme is that both thresholds are time-invariant. Deciding on the correct values is a hard problem and, more importantly, their optimal values depend heavily on the workload (Section 2). Under light I/O, the high threshold should have a large value in order to increase the write cache ratio. On the other hand, under heavy I/O, it is important for the system to maintain a sufficient number of available clean pages and a smaller value for the threshold is preferable.

We take a rate-based, adaptive approach to setting the high watermark, deriving its value from the rate at which the system dirties pages.

Let $h(t)$ be the value of the time-variant high watermark and $d(t)$ the rate that processes are dirtying new pages. This rate is measured by examining the number of “set dirty bit” operations at every time unit.

For an incoming I/O rate $d(t)$, the value for the high watermark at the end of the period is $h(t)$. If the data rate changes, we adjust the value of $h(t)$ based on the following intuition:

- If $d(t) \geq d(t-1)$ then the value of $h(t)$ should be reduced
- If $d(t) \leq d(t-1)$ then the value of $h(t)$ should be increased

Due to the computational requirements of performing complex arithmetic calculations in the kernel and the frequency of the operation, none of the advanced smoothing algorithms [2] are suitable for adoption. Control theory methods are also not practical because the watermark variance is not linear and depends heavily on the workload, making linear approximations hard [28].

We picked a simple smoothing function to adjust the high watermark value, based on the statistics we collect about the incoming data rate. Specifically, we use a formula for the relative change in the value of $h(t)$ so that the value of the threshold is inversely proportional to the change in the incoming data rate.

$$h(t) = h(t-1) \frac{d(t-1)}{d(t)}$$

We experimented with several different functions and algorithms to adjust the value of $h(t)$, such as moving average methods, step functions or exponential increase and backoff. This simple scheme provides competitive results along with very low computational requirements—an important factor given the frequency of this operation in the kernel. Nam and Park [19] provide some more mathematical insight to a similar scheme when describing a destaging algorithm for RAID-5 arrays. A detailed analytical study of the

watermark variance problem, part of future work, may highlight areas of improvement and potentially lead to optimizations to our framework.

We also take a similar adaptive approach to setting the low watermark, deriving its value from both the process writing rate and on the I/O rate that the storage device can sustain. The difference between the low and high watermark determines the amount of data to destage upon activation of the write-out process. The value of the low watermark $l(t)$ cannot be higher than $h(t)$ and it depends on the rate $c(t)$ the memory manager is able to clean pages, i.e. flush data to the device. If the device can sustain a high I/O rate compared to the incoming data rate then flushing can be less aggressive and $l(t)$ can have a higher value. On the other hand, if the incoming rate is higher than the outgoing rate, it is essential for the system to make clean pages available and $l(t)$ should be low. We define $l(t)$ as:

$$l(t) = l(t-1) \frac{d(t-1)}{d(t)} \frac{c(t)}{c(t-1)}$$

Again, we experimented with other scaling functions in adjusting the low watermark, but found that this simple scheme performed well in practice and has low computational requirements.

To avoid overtuning the watermarks, we rate limit the change of $h(t)$ and $l(t)$ so that $\frac{1}{2}l(t-1) \leq l(t) \leq 2l(t-1)$ and $\frac{1}{2}h(t-1) \leq h(t) \leq 2h(t-1)$.

The parameter t defines the time at which the system samples the I/O rates and sets the value of the watermarks. The value of the interval between two measurements of the incoming $d(t)$ and outgoing $c(t)$ rates is critical in how fast the system adapts to the workload patterns. Frequent measurements allow the system to adapt more quickly. On the other hand, sampling the page states too often increases computational overhead. We discuss the importance of the time unit selection in the experimental section.

3.2 A Read-Write Cache

The goal of a unified read-write caching scheme differs from that of a write-only cache. A write-only cache attempts to keep as many dirty buffered pages as possible without running out of available memory. A read-write cache must preserve a useful population of read-cache pages. Reserving more memory pages to buffer writes reduces cache hit rates, because it reduces the effective size of the read cache.

We define h_{max} as the maximum possible occupancy of memory with dirty pages before the write-back starts, so that the inequality $h(t) < h_{max}$ is always valid. We extend the destaging algorithm presented in the previous section by adaptively varying the

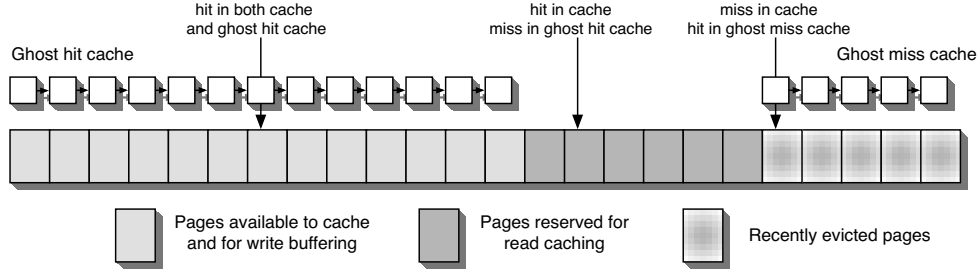


Figure 7: Using two ghost caches to identify the current working set

maximum occupancy ratio h_{max} . The intuition behind varying the h_{max} value is to provide an upper bound to the number of dirty pages in memory so that new dirty pages do not replace pages that are used for read caching by processes. Based on the fact that a read blocks the calling application whereas a write does not, our main goal is to maximize the read hit rate. To do so, we use a set of heuristics to identify what percentage of the RAM is actively used by processes to cache read data.

We use a ghost cache that holds meta-data information on blocks recently evicted from the cache (the ghost miss cache in Figure 7). In this way, we record the history of a larger set of blocks than can be accommodated in the actual cache. In the ghost miss cache, we keep an index of the blocks that were replaced as a result of write buffering only. If a clean block is replaced by another block, due to the regular eviction policy after a read, we do not add it to the ghost cache.

When a process issues a read, we look at the cache to determine whether it contains the requested block. If the block is not found, but it was recently replaced due to write buffering, its metadata information resides in the ghost cache. This indicates that if the system were buffering fewer write pages then this request would have resulted in a cache hit. An actual cache miss that hits in the ghost cache indicates that aggressive write buffering is interfering with read cache performance and that the value of h_{max} should be reduced. We note that our ghost cache does not rely on a specific eviction policy. It simply tracks recently evicted pages.

The ghost cache contains indexes of already evicted blocks, and, consequently, the effectiveness of the scheme is limited, because the signaling of over-buffering comes too late—the blocks must be fetched again from the storage device.

To make our scheme more proactive, we introduce a second ghost cache. For clarity reasons we call the first cache a **ghost miss cache** and the second a **ghost hit cache**. In contrast to the miss cache, the hit cache does not contain evicted blocks. Rather, the ghost hit

cache contains the contents of a smaller virtual memory (Figure 7). The ghost hit cache contains *all* of the write buffered pages and the most recent/frequent read cache pages. The memory area outside of the ghost hit cache contains the least recent/frequent read cache pages. As an extreme example, if all the h_{max} fraction of the available memory contained dirty buffered pages then all read cache hits would occur in the reserved area.

The ghost hit cache allows us to detect the potential negative effects of aggressive write buffering prior to incurring penalties from cache misses. If all read requests hit in the ghost hit cache then a smaller amount of memory would capture the current working set and more write buffering can be allowed. Alternatively, read requests that are cache hits but are misses in the ghost hit cache indicate that further write buffering, shrinking the available read cache, will probably result in reduced cache hit rates, because the effective memory space for read caching is running out. Read cache hits both inside and outside the ghost hit cache scale the amount of memory available for write buffering and read caching based on the current memory usage and workload.

Our ghost hit cache works for all modern read caching algorithms that maintain recency and/or frequency queues. Our implementation derives the ghost hit cache and reserved regions from Linux’s two queue approximate LRU implementation, which identifies the pages that would be evicted were the memory smaller. Many other recency and frequency based caching algorithms [11, 12, 17, 20] make similar information available.

We vary the value of $h_{max}(t)$ based on the hit rates of the two ghost caches as follows:

$$h_{max}(t) = h_{max} \left(1 - \frac{C(t) - GH(t) + GM(t)}{reads(t)} \right)$$

in which $C(t)$, $GH(t)$, and $GM(t)$ are the number of hits in the page cache, ghost hit cache, and ghost miss cache, $reads(t)$ is the number of total read requests during the last time interval respectively. The quantity $C(t) - GH(t) + GM(t)$ counts the read requests that

fall into the reserved area and in recently evicted pages. A large fraction of read requests falling in these regions indicates that aggressive write buffering is consuming memory in the ghost hit cache needed for read caching or that the current working set is larger than the available memory in the ghost hit cache.

We note that $h_{max}(t)$ is always lower than h_{max} . $h_{max}(t)$ reduces the high threshold based on the distribution of reads in the previous time period only. If there are no read requests, the value defaults to h_{max} . Thus, the initial value h_{max} should indicate the highest possible amount of RAM that the system administrator wants to devote to buffering. To avoid pathological cases that arise when over-tuning $h_{max}(t)$, we rate limit the change so that $\frac{1}{2}h_{max}(t-1) \leq h_{max}(t) \leq 2h_{max}(t-1)$.

Our implementation sets the ghost hit cache size to be an h_{max} fraction of the available memory. The ghost miss cache keeps an index of the blocks that would remain in the cache if the amount of available RAM were larger by 20%. The size of the ghost caches affects the responsiveness of the system in case of cache misses. In a small ghost cache, a few missed reads will force write buffering to be less aggressive. On the other hand, a larger cache requires a higher number of misses to limit write buffering.

The value of the static h_{max} parameter should depend on the relative cost of reads and writes in the system. In a RAID-5 system in which writes are expensive, more buffering space will improve performance. On the other hand, if a log-structured file system is used, more space should be devoted to read caching. In the results section we provide more information about the importance of parameter selection.

Finally, it is important to differentiate between the h_{max} threshold and the memory pressure point at which all writes become synchronous until enough pages are freed. Our framework does not stall writes even if they occupy more than h_{max} percent of the cache size, it just mandates more aggressive write-back. In the AWOL implementation, we define the memory pressure point $h_{pres} = \frac{CacheSize - h_{max}}{2}$. If the number of dirty pages reaches h_{pres} buffered writes become synchronous.

4 Opportunistic Queuing

Our adaptive high-low watermark algorithm attempts to optimize the write-out of dirty pages by deciding when to start the destaging process (high watermark) and how much data to flush (low watermark). Deciding what to destage is another important factor that affects the system performance. For example, it is preferable to flush a buffer that is physically close to a stream of read requests currently being serviced by the disk. This kind of optimization is not feasible

in the memory, because the memory manager has no knowledge of file system and device characteristics, e.g. the device LBN corresponding to a dirty page. Our framework addresses this problem by delegating the responsibility of selecting which pages to be cleaned to the I/O scheduler. We achieve this by introducing a unified memory-scheduler queue. We have implemented our modifications, valid for local file systems only, in Linux. However they apply to most operating systems.

A typical I/O scheduler differentiates between synchronous (read) and asynchronous (write) requests by using two separate sets of queues for each type of request. Synchronous requests have a short deadline, on the order of microseconds, so that requests are dispatched quickly and the application does not block waiting for the operation to complete. On the other hand, asynchronous operations have less strict deadlines, on the order of a few milliseconds, depending on the scheduler's policy.

In both queues, the I/O scheduler keeps the list of pending I/O requests sorted by logical block number. When a new I/O request is issued, it is inserted into the LBN sorted list of pending requests. This prevents the drive head from seeking all around the disk to service I/O requests. Instead, by keeping the list sorted, the disk head moves in a straight line around the disk. If the hard drive is busy servicing a request at one part of the disk and a new request comes in to the same part of the disk, that request can be serviced before moving off to other parts of the disk.

We enhance the scheduler by adding a third queue for pages, namely the *opportunistic queue*. This queue maintains an LBN sorted list of pages that are in memory and have not yet been submitted to the device for write-back. When an application issues a write, the kernel marks the buffers dirty and, at the same time, it places a pointer to the buffers in the opportunistic queue. In contrast to the conventional scheduler queues, requests in the opportunistic queue do not have an assigned deadline. Requests may remain in the queue until the memory manager dispatches the corresponding page to the storage layer. Then, the buffer is moved to the asynchronous list and its priority is determined by the scheduling algorithm.

When the scheduler is ready to dispatch the next request from the pending queue, it searches both the asynchronous and the opportunistic list for requests that are close to the one that is about to be issued. Figure 8 illustrates this process—it represents a single queue at the device for simplicity. The spatial criterion for proximity follows the same policy that current schedulers use and is based on the logical block number (LBN). Head positioning information and knowledge about the physical layout on the disk are not available

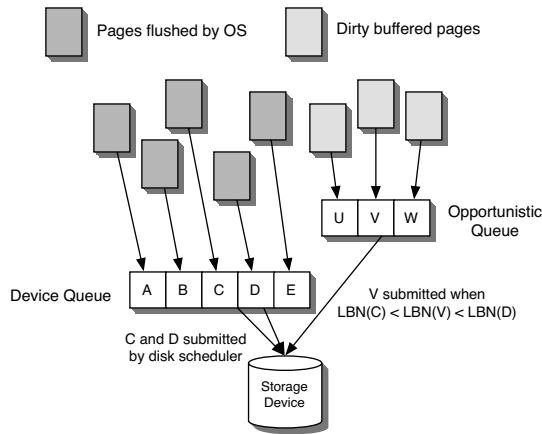


Figure 8: Opportunistic queuing

at this layer. Prior studies have proved the efficacy of a scheduling scheme based on LBNs [8, 15].

If one or more proximal requests are found, the I/O scheduler adds them into the dispatch list and removes them from the opportunistic queue. After the I/O is complete, it notifies the memory manager that the corresponding page is clean, so that the dirty bit is removed. If no appropriate request is found in the opportunistic list, the scheduler moves on to the next request in the synchronous queue. There is no performance penalty when no dirty buffered pages are dispatched; the opportunistic queuing scheme is an optimization, requiring only a few bytes of extra memory to keep the index of the buffer heads.

We implemented opportunistic queuing by modifying the deadline scheduler. Note that, this optimization requires modification to the data structures and logic of the scheduler but makes no assumption about the scheduling algorithm itself. Therefore, this mechanism is applicable to I/O schedulers in general and to the other Linux I/O schedulers in particular. In order to decide which requests to consider proximal, we select a simple criterion: its LBN must lie between those of the two next requests to be dispatched to the device.

The opportunistic queuing mechanism allows the system to commit pages to stable storage prior to them being submitted for write-back. This minimizes the interference of reads and writes at the disk level. The unified memory/scheduler queue reduces the average time for writes by ordering requests to the disk so that the service time for each request is minimized.

5 Evaluation

We have implemented our proposed changes in the 2.6.21 Linux kernel. We ran the experiments on a dual-core Xeon® machine with 2GB of RAM out of which about 1.5GB can be used for the page cache.

The rest of RAM is reserved for the operating system itself and for running applications. For experiments on a local file system, we used a dedicated SATAII-300 hard drive. To evaluate our framework in a heterogeneous environment, we also performed measurements over the Network File System (NFS) using two different networks: gigabit Ethernet and 10-Gbps Infiniband. For Infiniband, we measured the performance of NFS/RDMA: a high-bandwidth, low-latency setup. The NFS server has 8GB of RAM and exports the filesystem asynchronously. By performing memory-to-memory operations (NFS client to NFS server), we measured the performance of the memory optimizations without the bottleneck of the disk device. Finally, throughout the experiments, we adjusted the value of the watermarks every one second. We experimented with different values of this parameter later in this section.

In order to evaluate our solution, we performed a series of microbenchmark and macrobenchmark experiments. The first set of experiments were based on IOzone: a popular benchmark suite that measures throughput and latency of I/O operations.

We use the following IOzone microbenchmarks:

- **No Reader, Sequential Writer (NRSW):** One process writing to a 1GB file sequentially.
- **No Reader, Zipf Writer (NRZW):** One process writing 1GB worth of data according to a Zipf distribution. Thus, certain popular blocks receive many accesses.
- **No Reader, Variable Writers (NRVW):** Several IOzone clients executing the NRSW or the NRZW workloads at random intervals. Each process uses a file between 100MB and 512MB in size. There are always between five and eight processes running in the system.
- **Sequential Reader, Sequential Writer (SRSW):** Two processes reading and writing different 1GB files sequentially.
- **Random Reader, Random Writer (RRRW):** Two processes reading and writing different 1GB files randomly.
- **Zipf Reader, Zipf Writer (ZRZW):** Two processes reading and writing different 1GB files according to a Zipf distribution.
- **Variable Readers, Variable Writers (VRVW):** Several IOzone clients executing the SRSW or the ZRZW workloads at random intervals. Each process uses files between 100MB and 512MB in size. There are always between five and eight processes running in the system.

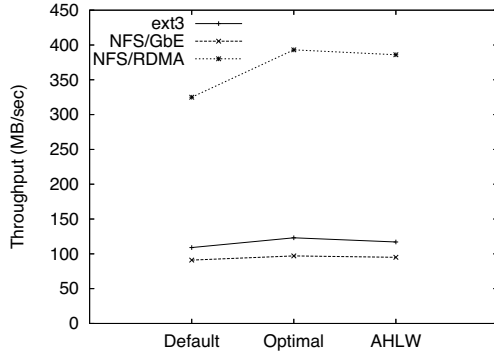


Figure 9: Throughput of a sequential writer (NRSW)

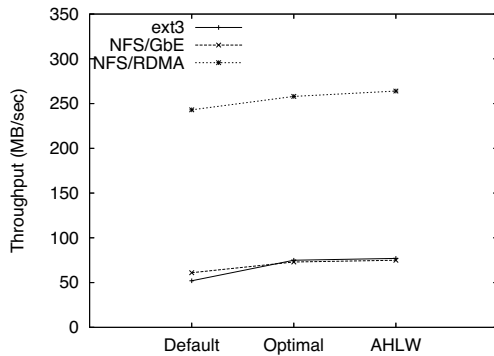


Figure 10: Throughput of a Zipf-distribution writer (NRZW)

5.1 The Adaptive High-Low Watermark Algorithm

First, we evaluate the effectiveness of the adaptive high-low watermark (AHLW) destaging algorithm using the IOzone workloads that perform writes only (NRZW and NRSW). These experiments do not use our read-write caching or scheduling optimizations. Figure 9 shows the throughput of Default, Optimal and AHLW under a sequential write (NRSW) workload. Default is the mainstream kernel with the default settings for deciding when to start the writeout (10% of available RAM). In Optimal, we have modified the value of the `dirty_background_ratio` to the optimal setting for this particular workload, which is 1%. We derived this number by manually altering the value of the parameter and rerunning the experiment until the highest throughput has been reached. Finally, AHLW is our modified version of the Linux kernel. AHLW performs almost as well as the optimal setup for the Linux kernel.

Figure 10 compares the throughput of Default, Optimal, and AHLW but this time under a workload that exhibits many rewrites (NRZW). AHLW throughput is slightly higher than Optimal—the optimal value of the `dirty_background_ratio` in this experiment is 41%. This is because the default Linux kernel uses a single threshold as opposed to the double watermark config-

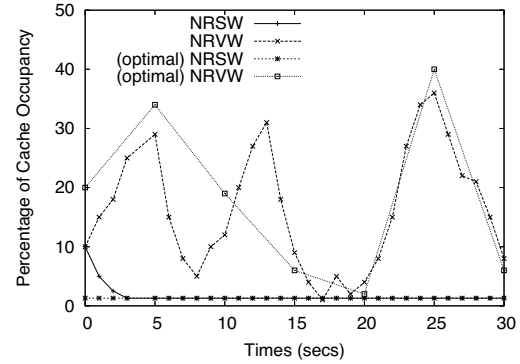


Figure 11: Variance of the high watermark for the NRSW, NRWV workloads and comparison with the optimal watermark values

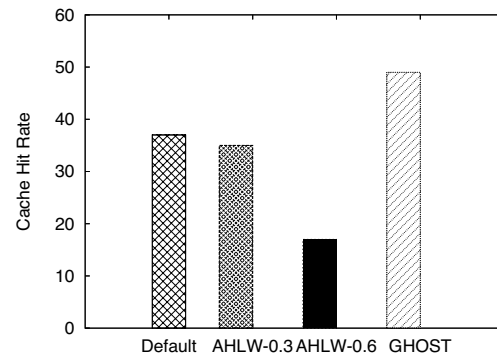


Figure 12: Comparison of cache hit rates for Default, AHLW and GHOST (ZRW)

uration. The double threshold decouples the decision of when to destage pages from how many pages to destage.

We now examine the performance of the weighting function that controls the high watermark. For the NRSW and NRWV benchmarks we plot the value of the watermark along with the optimal watermark value. We computed the optimal value by manually adjusting its value, rerunning the experiment from each time point and examining the number of destage I/Os, for each value. This optimal value is dynamic, whereas Figures 9, 10 use a static optimal. Figure 11 shows that in the case of a steady-rate workload (NRSW), the adaptive watermark quickly converges to its optimal value. For the variable rate workload (NRWV), the figure shows that there is room for improvement in the AWOL framework for a non-static workload.

5.2 The AHLW Algorithm with Ghost Caching

We now concentrate on the effect of write buffering on the cache hit rate. Specifically, we measure the system throughput and the cache hit rate under a workload that includes many re-reads and re-writes

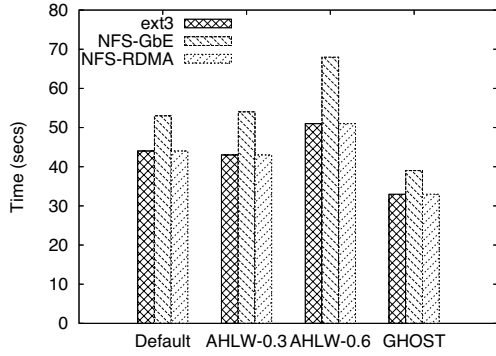


Figure 13: Execution times of Default, AHLW and GHOST under a read-write workload (ZRZW)

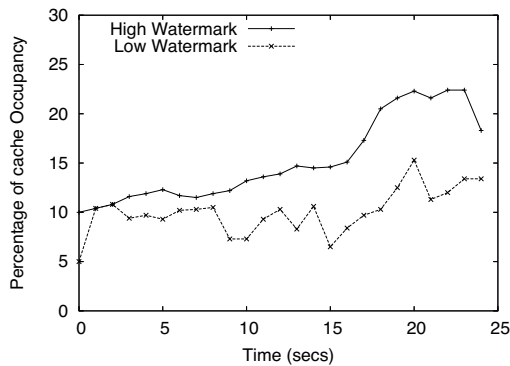


Figure 14: Variance of the high and low watermarks in GHOST (ZRZW)

(ZRZW). We compare Default, AHLW with $h_{max} = 0.3$, AHLW with $h_{max} = 0.6$, and GHOST: the Linux kernel enhanced with both the adaptive watermarks and the ghost caching optimizations. For AHLW the h_{max} value is static.

In this scenario, the cache hit rate affects the performance dramatically. Figure 12 plots the cache hit rate for each of the configurations. GHOST achieves the highest hit rate. In contrast, AHLW with $h_{max} = 0.3$ yields the lowest hit rate but provides the fastest writes (not shown). Overall, GHOST provides the best performance. The slower the device the more evident are the advantages of ghost caching (Figure 13).

Figure 14 shows the variation of the high and low watermarks in GHOST as a function of time for the last experiment. Due to block reuse (cache hits), the rate of incoming I/O requests is not constant and the watermarks increase and decrease. Also, the sudden rises and drops in the value of $h(t)$ are due to the h_{max} constraint imposed by the ghost caching algorithm. The value of the low watermark shows more instability due to the non-constant rate that the device sustains for random writes and reads.

Lastly, Figure 15 shows the read throughput of

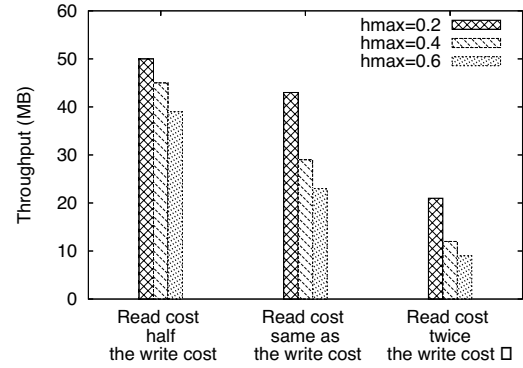


Figure 15: Comparison the read throughput as a function of the relative read-write cost (ZRZW)

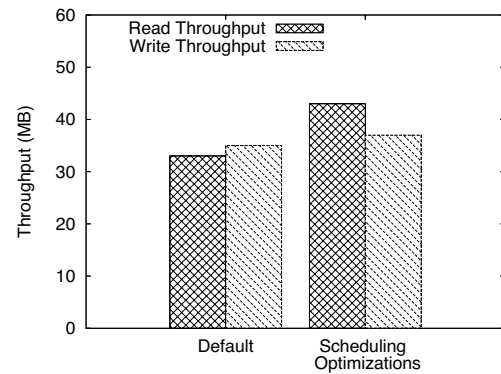


Figure 16: Comparison of the read and write throughput with and without opportunistic queuing (RRRW)

GHOST in the ZRZW benchmark for different values of the h_{max} parameter as a function of the disk read/write cost. In systems where disk writes are more efficient than reads, *e.g.* log-structured designs, it is important to maintain a high cache hit ratio, and, hence, the value of h_{max} should be relatively low. In contrast, in systems where writes are expensive, *e.g.* RAID-5, the system should allow for more buffering. For this experiment, we artificially delay its type of request in the kernel to simulate the different environments. Our adaptive scheme adjusts the $h_{max}(t)$ value to the experienced workload. In general, the h_{max} parameter should be set to a high value (greater than 0.4 of the available memory).

5.3 I/O Scheduler Optimizations

We now assess the effectiveness of the I/O scheduler optimizations using the RRRW workload. The Linux kernel enhanced with the opportunistic queuing mechanism performs 20% fewer disk operations than the unmodified kernel. As a result, throughput is improved by more than 35% for reads (Figure 16). Random writes that are proximal to reads are scheduled immediately.

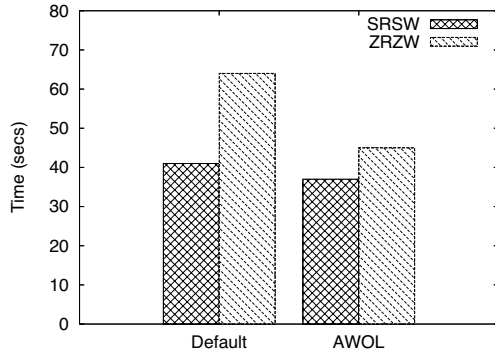


Figure 17: Performance of the ZRZW and SRSW benchmarks for Default and AWOL

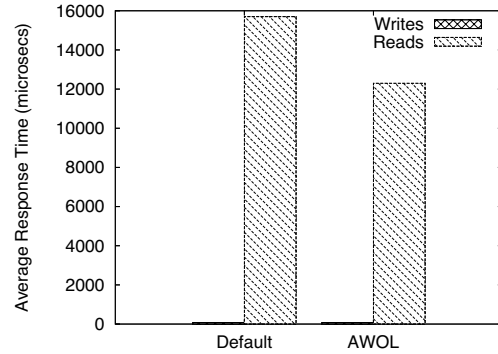


Figure 19: Average response times with a variable I/O rate (VRVW)

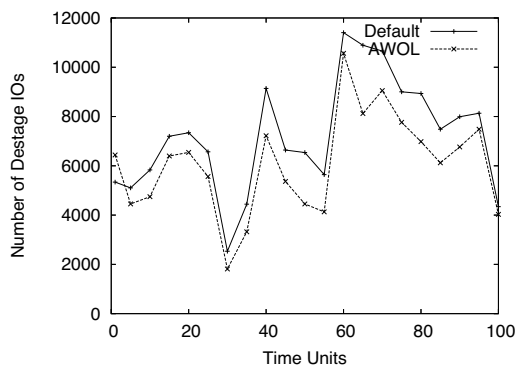


Figure 18: Number of destage I/Os per time unit as a function of time (VRVW)

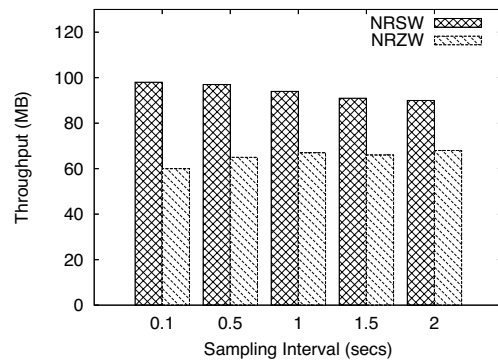


Figure 20: Throughput comparison under different values of the sampling interval (NRSW, NRZW)

This reduces the contention for disk bandwidth between reads and asynchronous write operations. Write throughput increases only modestly, because all write operations are asynchronous.

5.4 Putting it All together

We compare the performance of AWOL, our complete framework, with the unmodified Linux kernel. First, we compare the execution times of the SRSW and ZRZW workloads in ext3. Figure 17 shows that AWOL yields close to 35% improvement when compared with unmodified Linux for Zipf-distributed reading and writing (ZRZW). The performance improvement for sequential reading and writing (SRSW) is close to 10%. For SRSW, the superior performance arises from scheduler optimizations alone, because there are no potential benefits from read caching.

In the next experiment we examine AWOL's performance under a variable rate (VRVW) workload. Figure 18 plots the number of I/O destages (reads or writes) per time unit as a function of the time. The rises and drops in the graph show the changing data rate. A higher rate leads to more I/Os being dispatched to the device. AWOL performs fewer destage I/O operations

than Linux on average. Figure 19 shows the user-perceived response times for the same experiment. All writes are buffered and complete in microseconds. Higher cache hit rates (ghost caching) and more efficient I/O scheduling (opportunistic queueing) result in much shorter average read response times for AWOL.

5.4.1 Adjusting the Sampling Frequency

Finally, we examine the importance of the sampling frequency. Figure 20 shows the throughput of the system under different values of this parameter for the NRSW and NRZW workloads. For the sequential writer example, frequent measurements allow the system to reach the optimal watermark value faster, at a price of higher CPU consumption (Figure 21). For a Zipf writer, too frequent measurements are prone to overtuning the watermarks, which results to lower throughput. For a read-write workload (ZRZW) with many re-reads, the cache hit rate is not affected by the sampling frequency (Figure 22). As with other system parameters, the optimal value of the sampling period depends on the experienced workload. In practice, a value of 1 second provides good throughput along with low computational requirements.

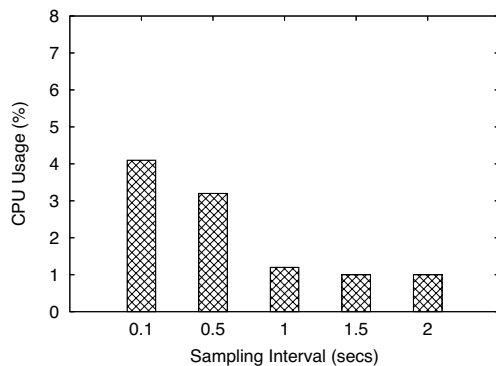


Figure 21: CPU consumption for AWOL under different values of the sampling interval (NRSW)

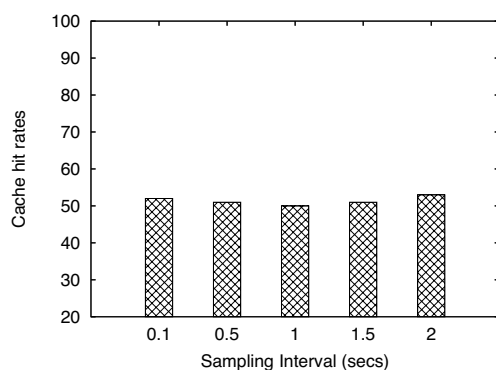


Figure 22: Cache hit rates for AWOL under different values of the sampling interval (ZRZW)

5.5 Macrobenchmarks

In the next experiment, we measure the time to untar and compile the Linux 2.6.20 kernel. This real-world workload exhibits lots of overwrites and intermediate, short-lived files. It accesses data sequentially (for the most part) with reads and writes interleaved. We compare the performance of Default and AWOL. The scheduler and destaging algorithm optimizations enable AWOL to reduce the execution time by almost 21% when compared with Default. Ghost caching also has some effect on this experiment. The cache hit rate (for both read and writes) is improved by 12%.

We also run TPC-C [26], a data-intensive, online transaction processing benchmark, for approximately one hour on a Postgres database. TPC-C issues small 4 KB random I/Os, two thirds of which are reads. The metric for evaluating TPC-C performance is the number of transactions completed per minute (tpmC). The amount of RAM available is critical to the reported performance. In our case, RAM covers only 20% to 30% of the working set. Figure 23 shows the throughput of unmodified Linux relative to AWOL for ext3 and NFS-RDMA. Our results are normalized because the

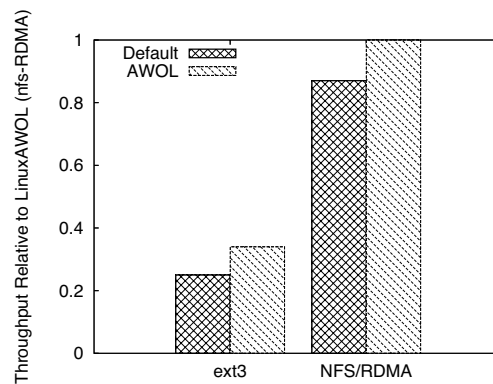


Figure 23: Performance of the TPC-C benchmark

test is unaudited. AWOL improves performance by more than 25%. The average value of $h(t)$ is 0.03 and its max value reaches 0.4. For the low threshold these values are 0.01 and 0.3 respectively. The performance difference of Default and AWOL comes as no surprise given the data-intensive nature of the benchmark.

6 Related Work

Early works that compare write-back caching observe that using cache memory for dirty pages may reduce read hit rates and identify that reads are more important than writes [13]. In fact, this was one of the fundamental arguments to continue using write-through caching.

Concerns about running out of available memory for dirty data, which results in slow write-back and stalls, arise first in processor caches, owing to their smaller sizes. Skadron [24] proposes the allocation of additional memory for dirty data and lazy retirement policies to mitigate these effects.

The periodic update policy used by most operating systems, e.g. every 30 seconds, leads to I/O bursts that can hamper system performance. Through analysis and simulation, Carson and Setia [5] showed that for many workloads periodic updates from a write-back cache perform worse than write-through caching. They suggest two alternate disciplines: (1) giving reads non-preemptive priority and (2) interval periodic writes in which each write gets its own fixed period in the cache. The first may starve writes indefinitely. The second requires complex queuing and timing mechanisms, but does stagger writes in time, assuming that pages are dirtied over time. Mogul [18] implements an approximate interval periodic write-back policy that staggers writes in time using a small (one second) timer. His evaluation shows that interval periodic writes reduce both response time and its variance.

Golding *et al* [9] propose to delay write-back until the system reaches an “idle” period. This reduces the

delays seen by reads by delaying competing writes until idle periods, possibly with the help of non-volatile memory. One aspect of AWOL defers write-back to the same effect. However, AWOL also writes more aggressively at times and adaptively chooses between deferring and aggressively writing pages.

The notion of write-performance dominating overall system performance has a long history in file and storage system design. We invoke the same scaling arguments as do other researchers. As processors increase in speed relative to disk or network throughput, they dirty pages faster than the storage subsystem can clean them. This makes write performance the overall system bottleneck. Ousterhout [21] invokes this argument in the original paper on log-structured file systems.

Non-volatile memory (NVRAM) offers one way to improve write-performance and tolerate write bursts by making data persistent without sending it to disk [4, 6, 22]. Because NV-RAM is more expensive than regular RAM and the read cache does not need to be persistent for correctness, NVRAM systems partition memory into a volatile read cache and a non-volatile write cache. Thus, they do not consider the balance of read and write pages in a shared memory. Again owing to its cost, NVRAM also tends to be deployed in server systems or in hybrid disk devices, whereas we focus on the adaptive allocation of memory within the operating system.

RAID controllers that use non-volatile memory for writes employ adaptive destaging policies that either vary the rate of writing [27] or the destage thresholds [19] based on memory occupancy and filling and draining rates. Such systems have quite different goals from ours, because cached writes are persistent. They wish to delay destaging data as long as possible. In contrast, operating system memories contain volatile writes and, thus, must destage data more aggressively consistent with operating system age thresholds.

Also for RAID controllers, Gill et al. [8] integrate recency into disk scheduling algorithms in order to aggregate multiple writes to data prior to destaging the data to disk. In the context of RAID controllers, writes are particularly expensive as they involve disk seeks among all disks in the RAID group. In our opportunistic queuing optimization (Section 4), we also employ information about page state into making write scheduling decisions. However, Gill et al. do so to delay writing pages that may be re-written and are in NVRAM. We use it to perform opportunistic writing when writes will be inexpensive because the disk head is already near the write location. Alonso and Santonja [3] also use recency of write access to defer writing data for pages written multiple times.

Free-block scheduling [16] describes a framework

for enhancing disk head utilization and throughput by interposing background reading/writing tasks into the request stream. The authors include write-back as one of many possible uses. The disk write discipline of AWOL's opportunistic queuing mechanism is similar in concept to freeblock scheduling. In fact, we could incorporate freeblock scheduling to implement a more sophisticated version of our queuing optimization that uses more accurate information about the position and activity of the disk head. In contrast to freeblock scheduling, AWOL changes the fundamental write-scheduling framework. Pages are grabbed out of the page cache and queued immediately before the operating system submits them for write-back. In addition, our simple implementation, based on LBN alone, provides good empirical results without the complexity of implementing freeblock scheduling outside of disk firmware [15].

Recent caching work has explored the adaptive allocation of memory between recency and frequency for read pages. Two queue (e.g. 2Q [12]) versions of LRU split the LRU queue into a lower queue for pages accessed once (recency) and a higher queue (frequency) for pages accessed more than once. Several papers (ARC [17], LIRS [11]) size these queues adaptively in response to workload shifts. They do not consider the allocation of memory for write pages.

None of the described research balances the population of read and write pages in a shared memory and dynamically allocates memory across these classes of pages. EMC's Enginuity [7] is the exception. This storage controller manages a global memory for reads and writes, allowing the region used for non-volatile writes to grow and shrink over time in response to workload shifts. No algorithms or details are given.

Li et al. [14] classify writes by type in order to better manage a second-tier read cache. Writes that correspond to dirty pages that are evicted from a first level cache are good candidates for caching at the second tier, whereas writes that periodically clean dirty pages are poor candidates, because those pages are likely still cached at the first tier. Our system also implicitly classifies writes but in different dimensions. We are concerned with whether writes are synchronous, blocking an application, or asynchronous. We do not use explicit hints.

Finally, ghost caching has been used quite extensively to track pages beyond the size of available memory. Megiddo and Modha in ARC [17] provide an overview of the use of shadow (ghost) caches in the many read-caching algorithms that balance multiple queues. Wong and Wilkes [29] use the technique for exclusive caching in a hierarchy of caches. The major difference is that AWOL maintains two ghost caches;

a larger cache to detect misses that would be hits in a larger cache (similar to previous uses) and a smaller ghost cache to detect hits that would be misses in a smaller cache. The latter technique allows AWOL to detect when increased write-back caching would degrade cache hit rates and prevents AWOL from consuming too much memory for dirty data.

7 Conclusions

In this paper, we demonstrate how the static write policies used by the memory manager do not adapt well to the variable workloads modern operating systems experience. Overly aggressive write buffering eliminates the effective space for caching and hurts performance. On the other hand, if the memory manager starts the destaging process too early, the background write load interferes with foreground reads.

Our modifications to the memory manager and I/O scheduler enable the system to automatically tune the destaging process, depending on the workload. Our framework minimizes the interference of read and write traffic at the device level and also maximizes cache hit rates.

We implemented our changes to the memory manager and I/O scheduler in the 2.6.21 Linux kernel. We will make these changes available to the Linux community prior to the publication of these results under the GNU General Public License [1], which means that the source code will be freely-distributed and available.

References

- [1] Gnu general public license. Version 2, Available at <http://www.gnu.org/licenses/gpl.html>. Accessed 12/4/2007.
- [2] Smoothing data. Chapter on Data Smoothing (Wolfram Research) Available at <http://documents.wolfram.com/applications/>. Accessed 12/4/2007.
- [3] M. Alonso and V. Santonja. A new destage algorithm for disk cache: DOME. In *EUROMICRO Conference*, 1999.
- [4] P. Biswas, K. K. Ramakrishnan, and D. Towsley. Trace driven analysis of write caching policies for disks. In *ACM SIGMETRICS*, 1993.
- [5] S. D. Carson and S. Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transaction on Software Engineering*, 18(1), 1992.
- [6] K. Chen, R. B. Bunt, and D. L. Eager. Write caching in distributed file systems. In *IEEE International Conference on Distributed Computing Systems*, 1995.
- [7] Enginuity algorithms: Dynamically optimizing performance. Whitepaper. Available at <http://www.emc.com/pdf/techlib/c1033.pdf>. Accessed 9/4/2007.
- [8] B. S. Gill and D. S. Modha. WOW: Wise ordering for writes—combining spatial and temporal locality in non-volatile caches. In *File and Storage Technology Conference*, 2005.
- [9] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness Is Not Sloth. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*.
- [10] The IOzone Benchmark. Available at <http://www.iozone.com>.
- [11] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS*, 2002.
- [12] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Conference on Very Large Databases*, 1994.
- [13] N. Jouppi. Cache write policies and performance. Technical Report WRL 91/12, Digital Western Research Lab, 1991.
- [14] X. Li, A. Aboulmaga, K. Salem, A. Sachendina, and S. Gao. Second-tier cache management using write hints. In *File and Storage Technologies Conference*, 2005.
- [15] C. Lumb, J. Schindler, and G. Ganger. Freeblock scheduling outside of disk firmware. In *Conference on File and Storage Technologies*, 2002.
- [16] C. Lumb, J. Schindler, G. Ganger, and D. Nagle. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Operating Systems Design and Implementation*, 2000.
- [17] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *File and Storage Technologies Conference*, 2003.
- [18] J. Mogul. A better update policy. In *USENIX Summer Technical Conference*, 1994.
- [19] Y. J. Nam and C. Park. An adaptive high-low water mark destage algorithm for cached RAID5. In *Pacific Rim International Symposium on Dependable Computing*, 2002.
- [20] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD*, 1993.
- [21] J. Ousterhout and F. Douglass. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1), 1989.
- [22] J. F. Pâris, T. Haining, and D. Long. A stack model based replacement policy for a non-volatile write cache. In *Conference on Mass Storage Systems*, 2000.
- [23] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(1), 1990.
- [24] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *Symposium on High-Performance Computer Architecture*, 1997.
- [25] The Open Group. The Single UNIX Specification. Available at <http://unix.org>.
- [26] The TPC-C Benchmark. Available at <http://www.tpc.org/tpcc>.
- [27] A. Varma and Q. Jacobsen. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Transactions on Computers*, 47(2), 1995.
- [28] M. Vidyasagar. *Nonlinear Systems Analysis, Second edition*. Prentice Hall., Englewood Cliffs, NJ, 2000.
- [29] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

Trademark Notice:

NetApp and the Network Appliance logo are registered trademarks and Network Appliance is a trademark of Network Appliance, Inc. in the U.S. and other countries. Linux is a registered trademark of Linus Torvalds. Xeon is a registered trademark of Intel Corporation. UNIX is a registered trademark of The Open Group. All other brands or products are trademarks or registered trademarks of their respective holders and should be treated as such.

TaP: Table-based Prefetching for Storage Caches

Mingju Li

University of New Hampshire
mingjul@cs.unh.edu

Swapnil Bhatia

University of New Hampshire
sbhatia@cs.unh.edu

Elizabeth Varki

University of New Hampshire
varki@cs.unh.edu

Arif Merchant

Hewlett-Packard Labs
arif@hpl.hp.com

Abstract

TaP is a storage cache sequential prefetching and caching technique to improve the read-ahead cache hit rate and system response time. A unique feature of TaP is the use of a table to detect sequential access patterns in the I/O workload and to dynamically determine the optimum prefetch cache size. When compared to some popular prefetching techniques, TaP gives a better hit rate and response time while using a read cache that is often an order of magnitude smaller than that needed by other techniques. TaP is especially efficient when the I/O workload consists of interleaved requests from various applications, where only some of the applications are accessing their data sequentially. For example, TaP achieves the same hit rate as the other techniques with a cache length that is 100 times smaller than the cache needed by other techniques when the interleaved workload consists of 10% sequential application data and 90% random application data.

Index Terms: RAID, prefetch cache, sequential stream detection, read caches, read-ahead hit rate, I/O performance evaluation, disk array.

1 Introduction

Storage devices have evolved from disks directly attached to a host computer and controlled by the host's operating system into independent and self-managed devices containing tens-to-hundreds of disks accessed by several computer systems via a network. Large and mid-size storage devices have sophisticated controllers that control when and how disk data are stored, retrieved, and transmitted. In addition to the disks and controllers, storage devices have high-speed cache memory. The most effective way of speeding up storage systems is to ensure that required data are already loaded in the cache from the disks when read I/O requests for this data arrive, and

to ensure that there is sufficient cache space for all write I/O requests to be written to cache immediately.

The storage caching technique determines how the storage cache space is allocated between read and write request data. The read request data stored in a cache are further classified as re-reference data, prefetch data, and old request data. The re-reference data are prior read I/O request data that are accessed often. The prefetch data are prefetched from the disks into the cache by the caching technique before I/O requests for these data arrive. The old request data are prior read I/O request data that have not yet been re-referenced. The caching technique makes decisions on how much space to allocate to re-reference, prefetch, and old request data.

A key factor underlying the success of a prefetching technique is the effectiveness of the technique that identifies sequential streams in the I/O workload, where a stream refers to the sequence of I/O requests corresponding to a particular application that are submitted by the application over a period of time. A stream is said to be sequential if the corresponding data are being read sequentially. Sequential stream detection is a difficult task since a storage system has no knowledge of the file systems or of the applications accessing its data. Consequently, a storage system has to identify sequential streams in its workload based solely on the addresses of I/O requests. Unfortunately, I/O requests from various streams are interleaved, so the outstanding I/O workload at any point in time shows little sequentiality, even when each stream accessing the storage device is sequential. As a result, a storage sequential stream detection technique must search past request addresses to see if an incoming I/O request is contiguous to any past I/O request. Thus, in addition to enabling re-reference hits, old I/O requests that remain in the cache facilitate sequential stream detection.

The key disadvantage of using the cache to detect sequential streams is that valuable cache space must be used to store old request data. Previous studies have

shown that only a small percentage of the total I/O workload displays a high degree of re-reference hits [33, 40] since storage caches are at the bottom of the cache hierarchy in computer systems. However, even when the locality-of-reference in the workload is too low to justify a large read cache, it is still needed to store the large number of old I/O requests required to detect sequential streams. This is an inefficient use of scarce cache memory, which typically constitutes 0.1% to 0.3% of the available disk space [16, 24].

We propose using a separate data structure, namely, a table of request addresses, to detect sequential streams and to adapt the prefetch cache size efficiently based on the I/O workload. We refer to the our proposed prefetching technique as **Table-based Prefetching (TaP)**. The TaP table is used to record request addresses and determine the *optimum prefetch cache size* for TaP. We define the optimum prefetch cache size of TaP as the cache size that is sufficient and necessary to obtain TaP's best achievable read-ahead hit rate for a given I/O workload. When an arriving I/O request misses in the prefetch cache, the TaP technique searches the TaP table to check if this new request's address is contiguous to any prior request addresses. If so, TaP flags this I/O request as belonging to a sequential stream and activates prefetching, else the TaP technique inserts the address of this request into the TaP table. TaP dynamically adapts the size of the prefetch cache depending on the fraction of sequential streams in the I/O workload. When the prefetch cache gets full and requests are thrown out, the request's address is inserted into the TaP table. In addition, a flag is set in the corresponding entry of the TaP table. This information is used by TaP to determine whether the size of the prefetch cache is too small for the current workload.

The key advantages of the TaP technique are as follows:

1. The optimum prefetch cache size can be determined efficiently.
2. Old I/O request data need not be stored, so the cache space can be judiciously shared between the write data, the prefetched data, and the re-reference data. (Note that the TaP table could also be used to identify missed re-references. This approach is not discussed here since it is not the focus of this paper.) Having a larger prefetch cache can improve the overall efficiency of a prefetching technique by permitting more data to be prefetched with a lower probability of the data getting evicted before being used.
3. The address tracking mechanism is not limited by the size of the cache, so more of history can be stored. As a result, there is a higher probability that

the TaP technique would be able to identify sequential streams when the overall I/O workload contains a mix of requests from several random and sequential streams.

4. The TaP technique is simple but effective, and has low implementation complexity—its simplicity is one of its strengths.

Our simulation study shows that the TaP technique gives the same or better hit rate and response time as the other prefetching techniques used for comparison, while using a smaller read cache.

The rest of the paper is organized as follows. Section 2 presents a classification of existing cache prefetch techniques and the related work. The motivation and design of TaP technique are explained in Section 3. Section 4 presents the experimental evaluation of the TaP technique. Section 5 presents the conclusion.

2 Sequential prefetching techniques and related work

A sequential prefetching technique consists of three modules. The *sequential detection module* deals with the detection of sequential streams in the I/O workload; the *prefetching module* deals with data prefetching details like how much data to prefetch and when to trigger the prefetch; the *cache management module* deals with preserving useful prefetched data until I/O requests for the data arrive. It is the cache management module that determines the size of the prefetch cache and the prefetch cache replacement policy.

Sequential detection module: The sequential detection module determines whether a missed I/O request is part of a sequential stream. It is an optional module in prefetching techniques. Some prefetching techniques use access patterns that are not based on sequentiality to predict what data to prefetch [10, 17, 27, 41], while other prefetching techniques prefetch sequential data without any analysis of access patterns [37, 38]. For example, the Prefetch-On-Miss technique activates sequential prefetching whenever an I/O request misses in the read cache without checking for sequentiality [35].

If the sequential detection module is present in a prefetching technique, then it is activated when an I/O request misses in the cache. Most current prefetching techniques use the cache to track addresses [18, 19] or predict future accesses based on previous access patterns [6, 12, 20] in the cache. Some techniques use offline information [23, 25, 26] and need to know future data access patterns for prefetching. Every I/O request

that misses in the cache activates the sequential detection module which searches the read cache for contiguous data. If contiguous data are found in the cache, then a sequential stream is detected and prefetching is activated for the detected stream. Henceforth, we shall refer to these **Cache based Prefetching** techniques as **CaP** techniques. There are several variations of the CaP technique. For example, instead of detecting a sequential stream the first time contiguous data are found in the cache on a miss, the detection technique could treat the missed I/O request as a potential start of a sequential stream. In this case, prefetching is not triggered immediately. Instead, a flag is set in the cache line where the missed I/O request is loaded. If this flagged cache line is found to be contiguous to yet another incoming missed request, a sequential stream is detected and prefetch is triggered.

There is very limited prior work on table based sequential prefetching techniques in storage devices. A table has been used in main memory hardware caches to keep track of data access patterns [8]. Mohan et al. [32] developed an algorithm with a stream table for processor caches which determines the spatial locality in an application's memory reference. The table saves stream information that has been detected. A table has been used in disk caches to predict sequential accesses [21]. The table stores time stamps associated with each entry in the cache and these time stamps are used in making prefetching decisions. A prefetching technique for networked storage systems called STEP [29] has been proposed in a most recent study. It uses a table for sequential access detection and prefetching. The table is maintained as a balanced tree and each entry records information for a recognized sequential stream or a new stream. While TaP also has a table, unlike these techniques, TaP uses the table to store different information (for example, the TaP table does not store time stamps and does not record information for recognized sequential streams).

While a lot of work has been done on workload prediction and prefetching in numerous areas such as processors [8, 9], web architecture [15], databases [12, 36], and file systems [7, 28], the characteristics of their workload are different from those of a storage system workload. In addition, access pattern prediction in those fields requires application or file system information while such information is not available to an independent storage system. For example, ZFS uses semantic information about file system to detect sequential streams [1].

Prefetching module:

The prefetching module of a prefetching technique is activated when an I/O request hits in the prefetch cache or when the sequential detection module identifies a new sequential stream. The prefetching module determines

how much data to prefetch. If data corresponding to several I/O requests are prefetched at a time, then prefetching is not triggered every time there is a hit. The prefetching module determines when to trigger a prefetch. Therefore, the prefetching module determines the efficiency of a cache prefetching technique once sequential streams are detected. For example, it would be sufficient to prefetch only one request at a time for a sequential stream with a low arrival rate, but the technique would have to prefetch more requests for a sequential stream with a high arrival rate. However, other factors must be considered too. For example, if the traffic at the disks is high, then prefetching should be triggered early enough for the prefetched data to arrive at the cache in time to result in hits. Gill and Bathen [18] developed a technique that determines the prefetching trigger point and the prefetching degree (*i.e.*, amount of data to prefetch) based on the workload intensity and storage system load. Several other papers have analyzed the prefetching degree based on various factors [11, 13, 35, 37, 38].

Cache management module:

The cache management module determines the cache replacement policy and the prefetch cache size. Typically, the prefetch cache is managed along with the rest of the read cache as a single unit. There is no separate space or replacement policy allocated for the prefetch cache. Instead, prefetched data are loaded into the single cache and treated just like regular data. When the cache is full, prefetched data are thrown out like the rest of the data depending on the cache replacement policy. Treating prefetched data like the rest of cache data is not necessarily a good idea. Patterson et al. [34] developed a cache module which contains three partitions based on hint information from the applications. Pendse and Bhagavathula [35] divided the read cache into fixed-size prefetch and random (including re-reference and old I/O request data) caches, and analyzed the prefetch cache replacement scheme.

ACME [4] and ARC [31] are two techniques that have different replacement policies for different cache portions. They use a virtual cache to manage their cache replacement policies. ACME maintains data in caches as objects and a set of virtual caches (*i.e.*, tables) is designed to keep past object header information for the distributed caches. Thus header information in each virtual cache is used to make decisions regarding replacement policies for the corresponding real cache. ARC separates the cache into two portions, one dedicated to the most recently-used and the other to the most frequently-used data. A cache directory (or table) is used for tracking the "recency" and "frequency" of past requests to change the size of both cache portions. The virtual caches in these two techniques are used neither for detection nor

prefetching of sequential streams. SARC [19] divides the read cache into prefetch and random cache lists and compares the relative hits in the bottom of the two lists to adapt the size of the lists dynamically. However, their sequential detection module is based on CaP, so they store some old data (although in two lists) for sequential stream detection (and re-reference hits). Compared to the techniques above, TaP is designed to use a table for lower level storage sequential detection, prefetching, and cache sizing in low level storage.

2.1 Prefetching technique classification

As mentioned earlier, the sequential detection module is an optional component of prefetching techniques. Based on the existence or lack of the sequentiality detection module and on when prefetching is triggered, the sequential prefetching techniques can be classified as follows.

1. **Always Prefetch (AP):** There is no sequential detection module and this technique always triggers a prefetch regardless of whether an I/O request hits or misses in the cache.
2. **Never Prefetch (NP):** There is no sequential detection module and this technique does no prefetching.
3. **Prefetch on Miss (PoM):** There is no sequential detection module and this technique prefetches every time an I/O request misses in the read cache.
4. **Prefetch on Hit (PoH):** Prefetch is triggered by a cache miss with some detection schemes, and then every I/O request that hits in the read cache causes a prefetch. This is the only class of prefetch techniques that has a sequential detection module. If the degree of prefetch is high (*i.e.*, data equivalent to several I/O requests are prefetched), then prefetching is not triggered upon every hit.

Based upon whether the sequential detection module is cache based or table based, the PoH techniques are further classified as follows.

- (a) CaP: The set of prior I/O request data stored in the read cache are searched to identify the start of a sequential stream. Most existing storage system prefetch techniques belong to this category.
- (b) TaP: The set of prior I/O request addresses stored in the TaP table are searched to identify the start of a sequential stream and to determine the optimum prefetch cache size. Both TaP in this paper and STEP [29] are newly developed techniques that belong to this category.

3 Design of the TaP technique

3.1 Motivation and goal

The design of the TaP technique is motivated by the following observations of lower levels of storage systems:

1. There is little value in caching old request data because the proportion of this data that will be re-referenced is small [33].
2. Most I/O workloads contain some sequential access patterns because file systems and storage systems try to manage data layout on disk devices such that data that are sequential in the application and file system space are also sequential in the disk address space. However, individual sequential patterns are interleaved with each other and therefore the aggregate I/O workload displays little sequentiality.
3. Although current middle or large storage systems have big caches and powerful controllers, their prefetching performance is poor. The study in [39] shows that the prefetching technique does not benefit the performance of the evaluated storage system when there are more than four sequential I/O streams since the prefetching technique does not recognize the interleaved sequential pattern in the workload. In addition, most well-studied prefetching techniques with advanced sequential detection schemes are designed for higher levels of computer systems. These are not suitable for storage systems because they need information from file systems or applications which is not available to storage systems.
4. Performance of a sequential prefetching technique is degraded if it uses an inefficient sequential detection module because of the following reasons:
 - (a) False positive detection errors generate unnecessary I/O traffic at the disks and increase the response time by considering random data as sequential. Moreover, valuable cache space is used to store useless data, thereby displacing correctly prefetched data that get evicted from the prefetch cache before they are used. The AP and PoM techniques are both likely to cause this problem if the I/O workload contains random streams or partly sequential streams.
 - (b) False negative detection errors decrease the hit rate and increase the response time by failing to identify sequential streams in the workload. The NP technique always faces this problem since it never prefetches. The CaP technique

faces this problem when the workload consists of a mix of random and sequential streams because, in this case, the history of request addresses is too short to record sequential patterns.

- (c) Correctly prefetched data from sequential streams could be evicted before the data can be used. This can occur if the prefetch cache gets full. For a given read cache size, AP, PoM and the CaP techniques are more susceptible to this problem since either they prefetch too much useless data (as in AP and PoM) or they store data from past I/O requests (as in CaP).

With the above observations in view, TaP is designed to detect, prefetch, and cache only sequential streams into its prefetch cache. Consequently, TaP is capable of identifying the minimal amount of data that should be prefetched and cached, and can therefore maintain the cache size at an optimal level. We consider a workload solely consisting of reads—the write workload is handled by the write cache.

At the heart of the TaP technique is the TaP table. TaP uses this table for two crucial functions: sequential stream detection and cache size management. The address of a request that is not found in the prefetch cache, is searched in this table. If it is not found in the table either, then assuming that the address is part of a new sequential stream, the address of the next expected request in this stream is recorded in the table. If the assumption turns out to be correct, then the address recorded in the table will be seen in the workload in the near future, and TaP will begin prefetching that stream when this occurs. As the table is populated with new addresses of potentially sequential streams, old addresses that have not led to a stream detection so far, are evicted on a FIFO basis. In this way, the table plays a key role in TaP’s ability to detect sequential streams.

The TaP table also plays a central role in maintaining the prefetch cache size at an optimal level. In addition to addresses of cache misses, addresses of requests that are evicted from the cache before they are hit are also inserted into the TaP table. These addresses are marked with a special flag, **replaceFlag**, in the table. TaP exploits the possibility that such pre-hit evictions may be the result of a smaller than optimal cache in the following way. If such flagged, evicted streams are soon re-detected by the detection method discussed above, then TaP rightly concludes that the cache is undersized and initiates a cache size increment. This upward movement of the cache size is balanced by the TaP Decrement Module (discussed below), which maintains a downward pressure on the cache size to prevent cache inflation.

Table 1: Important constants in TaP

Variables/constants	Usage
prefetchDegree	prefetch size
triggerOffset	when to prefetch
strideRange	sequential stream detection range
incrAmount	how much to increase prefetch cache size
decrAmount	how much to decrease prefetch cache size
measurementWindow	time window for hit rate measurement

In summary, an address that ends up in the table does so in one of exactly two ways. A request that misses in the cache and the table is inserted into the table. The address of a pre-hit eviction from the prefetch cache is also inserted into the table. Data that are cached do not have entries in the table.

3.2 TaP pseudocode

The TaP pseudocode is listed in Figure 1. The important constants used in the pseudocode are listed in Table 1. The first three constants are the inputs to the TaP algorithm provided by the system administrator. These affect the detection and the prefetching module. The variables **prefetchDegree** and **trigOffset** relate to the prefetching module, and determine how much data to prefetch and when to trigger a prefetch. The prefetching module is separate from the sequential detection module and is not the focus of TaP, so the current version of TaP uses constant values. However, a more versatile prefetching module can be incorporated into TaP and will improve the overall performance of TaP. The constant **strideRange** is used to specify the sequential stream detection range. When the TaP Table is searched, a hit within a stride range is considered (**TableHit**(req, **strideRange**)). The reason for searching within a stride range is that operating systems sometimes submit requests out of sequence. The last three parameters affect the cache management module. They should be chosen carefully by the administrator because they determine the tradeoff between performance and cache size economy. While the pseudocode implements the TaP table as a queue for ease of explanation, a hash table is a more appropriate data structure for the table. In addition, the TaP table bound derived below (Equation 4) justifies that the growth rate of the table size is sufficiently small. Therefore, the table search time is likely to be negligible. Moreover, the short table search time also guarantees that the controller-CPU cost is small since searching the TaP table is the CPU’s

<hr/> Function TAPCacheManage(<i>req</i>) <hr/> <pre> 1 <i>totalRequests</i>++; 2 if <i>req</i> ∈ Cache then 3 ProcessCacheHit(<i>req</i>); 4 <i>totalHits</i>++; 5 else 6 ProcessCacheMiss(<i>req</i>); 7 if <i>totalRequests</i> % <i>measurementWindow</i> == 0 then 8 if <i>HitRateStable()</i> then 9 DecrCacheSize(<i>decrAmount</i>);</pre> <hr/>	<hr/> Function HitRateStable <hr/> <pre> 1 <i>currHitRate</i> ← <i>totalHits</i>/<i>measurementWindow</i>; 2 if <i>currHitRate</i> − <i>prevHitRate</i> ≤ δ then 3 <i>stable</i> ← TRUE; 4 else 5 <i>stable</i> ← FALSE; 6 <i>prevHitRate</i> ← <i>currHitRate</i>; 7 <i>totalHits</i> ← 0; 8 return <i>stable</i></pre> <hr/>
<hr/> Function ProcessCacheHit(<i>req</i>) <hr/> <pre> 1 Serve <i>req</i> from Cache; 2 Evict <i>req</i> from Cache; 3 if <i>req.prefetchTrigger</i> == TRUE then 4 <i>startAddr</i> ← <i>req.addr</i> + 1 + <i>triggerOffset</i>; 5 Prefetch(<i>startAddr</i>, <i>prefetchDegree</i>, <i>triggerOffset</i>);</pre> <hr/>	<hr/> Function Prefetch(<i>startAddr</i>, <i>degree</i>, <i>trigOff</i>) <hr/> <pre> 1 <i>endAddr</i> ← <i>req.addr</i> + <i>degree</i> − 1; 2 for all <i>i</i> ∈ [<i>startAddr</i>, <i>endAddr</i>] do 3 if Cache is full then 4 <i>evictedReq</i> ← FIFOEvict(Cache); 5 TableFIFOInsert(<i>evictedReq</i>, TRUE); 6 Fetch data of <i>i</i> from Disk; 7 Insert <i>i</i> into Cache by FIFO; 8 <i>trigReq.addr</i> ← <i>endAddr</i> − <i>trigOff</i>; 9 <i>trigReq.prefetchTrigger</i> ← TRUE</pre> <hr/>
<hr/> Function ProcessCacheMiss(<i>req</i>) <hr/> <pre> 1 if <i>t</i> ← <i>TableHit</i>(<i>req</i>, <i>strideRange</i>) then 2 if <i>t.replaceFlag</i> == TRUE then 3 IncrPrefetchCacheSize(<i>incrAmount</i>); 4 Prefetch(<i>req.addr</i>, <i>prefetchDegree</i> + 1, <i>triggerOffset</i>); 5 else 6 Fetch <i>req</i> from Disk; 7 TableFIFOInsert(<i>req</i> + 1, FALSE); 8 Serve <i>req</i> from Cache; 9 Evict <i>req</i> from Cache</pre> <hr/>	<hr/> Function TableHit(<i>req</i>, <i>strideRange</i>) <hr/> <pre> 1 for any <i>r</i> ∈ [<i>req.addr</i>, <i>req.addr</i> + <i>strideRange</i>] do 2 if <i>r</i> ∈ TAPTable then 3 Remove <i>r</i> from TAPTable; 4 return <i>r</i>; 5 return NULL</pre> <hr/>
<hr/> Function FIFOEvict(<i>queue</i>) <hr/> <pre> 1 <i>h</i> ← dequeue(<i>queue.head</i>); 2 return <i>h</i></pre> <hr/>	<hr/> Function TableFIFOInsert(<i>req</i>, <i>flag</i>) <hr/> <pre> 1 if TAPTable is full then 2 FIFOEvict(TAPTable); 3 <i>entry</i> ← enqueue(TAPTable, <i>req</i>); 4 <i>entry.replaceFlag</i> ← <i>flag</i></pre> <hr/>

Figure 1: TaP pseudocode

biggest cost.

As shown in the pseudocode, when a new request arrives, it is handled by the `TaPCacheManage()` function. The TaP cache manager decides whether the request is part of an already detected sequential stream or if it should be recorded for future detection.

ProcessCacheHit() If the request generates a cache hit, TaP serves the request from the cache. TaP interprets this request as being part of an already recognized sequential stream and prefetches the next request in the stream. The previous request is evicted from the cache.

ProcessCacheMiss() If the request is not found to be in the cache, then its address is searched in the TaP table. If the request's address is found in the table, then this implies that two "consecutive" requests have been detected. TaP takes this as an indication of the start of a sequential stream and begins prefetching this stream. In addition, if the `replaceFlag` field of the request's entry in the table is set, then this entry must be the result of a pre-hit eviction from the cache. Therefore, TaP increments the cache size by `incrAmount`. (The Increment Module is discussed in further detail below.) If a request is not found to be in the table, then the address of the expected succeeding request is recorded in the table for detection in the future.

The TaP cache manager periodically monitors the cache size for inflation. During every period of time where `measurementWindow` requests arrive, the TaP cache manager maintains a count of the total number of cache hits accrued.

HitRateStable() At the end of this measurement period, the cache manager compares the *short term hit rate* in the current window to its value in the previous window. We define the short term hit rate as the ratio of hits to total requests in a measurement window. If the current and previous values are within some small additive constant δ of each other, then the cache manager concludes that the hit rate has been fairly stable. It takes this as an indication that the cache is adequately sized and might even be inflated. Therefore, it decreases the cache size by a preset amount equal to the `decrAmount`.

We next describe the rationale behind the TaP Increment and Decrement modules.

3.3 TaP cache size management modules

The degree of sequentiality of the I/O workload changes over a time period. The TaP table is a useful tool for

determining whether the prefetch cache size is too small for the current workload. The `replaceFlag` field of an entry in the TaP table is used for this purpose. The default value of the `replaceFlag` variable is false. When a prefetched request is evicted before a hit by the replacement scheme, the request's address is inserted into the TaP table with the `replaceFlag` set to true. Whenever there is a table hit, the `replaceFlag` is checked. If the flag is true, then the prefetch cache size is increased. Thus, entries that are reinstated into the TaP table from the cache are used to detect whether the cache size is too small.

While reinstated entries are a reliable indicator of cache space scarcity, a perfect indicator of cache size inflation is not obvious. The TaP cache manager uses a "downward pressure" approach to cache size deflation using the measured short term hit rate. The basic idea is that whenever the TaP cache manager observes the hit rate measured over some short term window of time to be stable (`measurementWindow`), it (pessimistically) assumes that the cache is slightly inflated and begins a gradual decrease of the cache size. The decrease continues so long as the hit rate remains stable. If the cache size falls below the optimal value, then the hit rate changes and this change prevents the TaP cache manager from decreasing the cache size any further. Moreover, a smaller than optimal cache size will lead to pre-hit evictions from the cache into the TaP table and re-insertions from the TaP table into the cache, which will quickly trigger an increase in the cache size back to the optimal value. Thus, as a result of the downward pressure from the decrement module, the cache size always rides close to the optimal value.

While the cache size oscillates around the optimal value when the workload is stable, the extent of these oscillations is small as seen in Figure 3. For example, at time 50000, the optimal value for the cache size is 50. The TaP cache manager maintains the cache size close to this value with an oscillation of less than five cache lines independent of the optimal cache size. In addition, these oscillations do not burden the CPU much, because there are only a few more operations (such as increasing or decreasing the cache size) added in each of the `measurementWindows` where the oscillations occur.

3.4 Bounding cache and table size

The table size, T , refers to the number of request addresses that can be stored in the TaP table. The table replacement scheme is a FIFO policy. When a request address gets a hit, it is removed from the TaP table. The cache size, C , refers to the number of cache lines assuming that exactly 1 prefetch request is stored in each line. Without loss of generality, it is assumed that the prefetch

degree is 1 request and prefetch is triggered upon every hit in the prefetch cache.

Below we derive a simple optimum bound for the cache size and simple pessimistic bounds for the cache and table size. The TaP technique initializes the prefetch cache size to the optimum bound, since TaP continually adapts the size of the prefetch cache size to match the sequential degree of the workload. The TaP table size is set to the pessimistic bound since the memory space used by a table could be orders of magnitude smaller than the cache size (*i.e.*, a cache line is an order of magnitude larger than a table entry).

The prefetch cache size must be large enough to hold prefetched data from each of the sequential streams. Suppose there are S sequential or partly sequential streams accessing the storage device. Then the cache size must contain at least S lines.

$$C \geq S \quad (1)$$

We now derive a pessimistic bound for C (and T). For real storage systems, it is difficult to get information about the degree of sequentiality of each workload stream or the variance in the inter-arrival rate of each stream. So, we derive worst-case bounds using only the number of (sequential + random) workload streams M and the number of sequential streams S . The bounds are derived as a function of a parameter ϵ which represents the acceptable percentage of reduction in the read-ahead hit rate. That is, if the acceptable percentage reduction in the hit rate is given, then a pessimistic bound for C and T can be computed.

Consider a workload consisting of M interleaved streams. A prefetching technique tries to ensure that a request prefetched for stream i survives in the cache until the next I/O request from stream i arrives. Between two requests from stream i , there can be several requests from the $M - 1$ other streams. Of these $M - 1$ streams, there can be at most $S - 1$ sequential streams. A prefetching technique should ensure that the request from stream i is not evicted from the cache due to cache insertions resulting from these sequential stream arrivals. Let $seqarrival\#$ represent the number of requests that arrive from other sequential streams between two requests from stream i .

Consider a synthetic workload in which (a) future request arrivals are independent of past arrivals, and (b) there is equal probability that the next arrival is from any of the M streams. Then,

$$Pr(seqarrival\# = n) = \left(\frac{S-1}{S}\right)^n \times \frac{1}{S}$$

Some of these $seqarrival\#$ arrivals could be from rec-

ognized sequential streams and would hit in the cache. Therefore, the prefetches initiated by these requests would not need new insertions into the cache. The sequential arrivals that miss in the cache (and hit in the TaP table) are the only arrivals that cause new insertions into the cache. Let $insertions\#$ represent the number of requests that result in new insertions into the prefetch cache between two arrivals from a stream i . Hence, $insertions\# \leq seqarrival\#$.

Since the cache replacement scheme is a FIFO, a new prefetched request is stored at location 0 of the cache. For this prefetched data to be useful, its corresponding I/O request must arrive within C or fewer requests. The probability that the number of cache insertions in the worst case are no more than can fit in the cache (without evicting the prefetched request) is:

$$\begin{aligned} Pr(insertions\# < C) &= \sum_{k=0}^{C-1} \left(\frac{S-1}{S}\right)^k \times \frac{1}{S} \\ &= 1 - \left(\frac{S-1}{S}\right)^C \end{aligned}$$

Therefore,

$$Pr(insertions\# \geq C) = \left(\frac{S-1}{S}\right)^C \quad (2)$$

Equation (2) provides the probability that a prefetched request is evicted from the cache before it is used. We bound this probability of eviction to some small value $\epsilon > 0$.

$$\left(\frac{S-1}{S}\right)^C \leq \epsilon.$$

This implies that the cache size

$$C \geq \frac{\log(\epsilon)}{\log\left(\frac{S-1}{S}\right)}. \quad (3)$$

Following an approach analogous to the one taken for the cache size, we can obtain a bound on the TaP table size

$$T \geq \frac{\log(\epsilon)}{\log\left(\frac{M-1}{M}\right)}. \quad (4)$$

Although these bounds are derived and used for synthetic workloads, they are also a guide for evaluating TaP's performance on real workloads. The bound in Equation (4) allows us to choose the tradeoff between maximizing the hit rate and minimizing the table size in inverse relation to the value chosen for ϵ . The CPU cost incurred by TaP is dominated by the size of the table that is searched for hits. Since this cost is only logarithmically related to the miss probability and inverse-logarithmically related to the fraction of interleaving streams, it is not prohibitively large.

Table 2: Storage simulator setup

Disksim parameter	Value
storage cache line	8 blocks
prefetch cache replacement policy	FIFO
storage RAID organization	RAID 5
stripe unit size	8 blocks
number of disks	4
disk type	cheetah9LP
disk capacity	17783240 blocks
mean disk read seek time	5.4 msec
maximum disk read seek time	10.63 msec
disk revolutions per minute	10045 rpm

4 Experimental evaluation

We evaluate the TaP technique using the DiskSim 3.0 simulator [5]. Table 2 gives the setup used for our experiments. We configured four Cheetah9LP 9GB disks as a RAID-5 system. The cache is divided into cache lines of size 8 blocks with 512 bytes per block. The cache size and the I/O workload are varied in our experiments. We use both synthetic workloads and realistic workloads. The synthetic workload uses several possible combinations of random and sequential streams in order to evaluate the technique under different conditions. It should be noted that due to memory and computing constraints, our simulation storage setup is much smaller than real storage systems, so the workload is also scaled down appropriately.

We compare the TaP technique against the CaP, AP, PoM, and NP techniques. The storage system’s mean response time and the cache’s prefetch hit rate are measured for the various prefetch techniques. Parameters such as prefetching degree and prefetching trigger are set at similar values for each of the techniques. The prefetching degree is set at 1 (*i.e.*, only 1 request is prefetched), so prefetching is triggered upon every hit in the prefetch cache. The TaP table length is set at the upper limit for the workload (Equation 4). The memory space utilized by TaP in our experiments is negligible compared to the cache size—the maximum space used by the TaP table in all our experiments is 4KB. To ensure fairness, we compare the performances of the various techniques under similar workloads and cache sizes. Note that the cache size for compared techniques is set to the sum of the cache and table size used by TaP. Both TaP and CaP initiate prefetch under similar conditions—for TaP, prefetch is initiated upon the first hit in the table, and for CaP, prefetch is initiated when an incoming request is found to be contiguous to an old request stored in the read cache.

4.1 TaP cache size manager in action

The first experiment evaluates the performance of TaP’s cache size manager as the workload changes. Figure 3 shows the result of a simulation of the TaP cache manager when the synthetic workload illustrated in Figure 2 is used. The workload starts with 10 streams with a sequentiality of 10%. These short-lived streams can be seen as a dense band of mostly random points from time 0 to 10000 in Figure 2. At time 8000, 50 completely sequential streams arrive. These appear as almost horizontal lines from time 8000 to time about 140000 in Figure 2. The TaP cache manager reacts to the influx of sequential streams by increasing the cache size. When the 10% sequential streams finish, the TaP cache manager decrements the cache size, without opposition from the increment module, until the optimal size of 50 is reached. This size is optimal because there are only 50 sequential streams in the workload at this point. Approximately at time 80000, 100 completely sequential streams are added to the workload which prompts the TaP cache manager to increment the cache size to the new optimal value of 150. At time 140000, the first 50 sequential streams finish. Again, the downward pressure meets no resistance and the cache size settles to the optimal value of 100. At time 200000, 50 streams with sequentiality 70% arrive and the cache size is incremented to accommodate them. The increase is larger than 50 because more cache space is required to get hits on streams with lower sequentiality: this is because of the single unavoidable extra prefetch at the end of a sequential run in a partially sequential stream. The sequential streams that arrived at time 80000 finish at time close to 220000 and the 50 streams with 70% sequentiality finish a little after time 250000. Both of these events allow the decrement module to gradually decrease the cache size. The workload changes again at times 250000 and 280000 when 10 streams with sequentiality 90% and 20 streams with sequentiality 100% are added, respectively. The cache is still inflated when these streams arrive, and so the gradual decrease of the cache size continues. At time 300000, the cache stabilizes to a value optimal for the 90% sequential streams. When these end, the cache size finally decrements to the optimal value of around 20 for the last remaining 20 completely sequential streams. Figure 4 shows that on average, a hit rate close to the maximum achievable with the current workload, is maintained throughout the simulation.

In summary, this experiment illustrates that the TaP cache manager is appropriately responsive to the changes in a non-stationary workload. The increment module, using the pre-hit eviction information from the table, is highly effective in quickly incrementing the cache size to a value that is optimal for the current workload. The

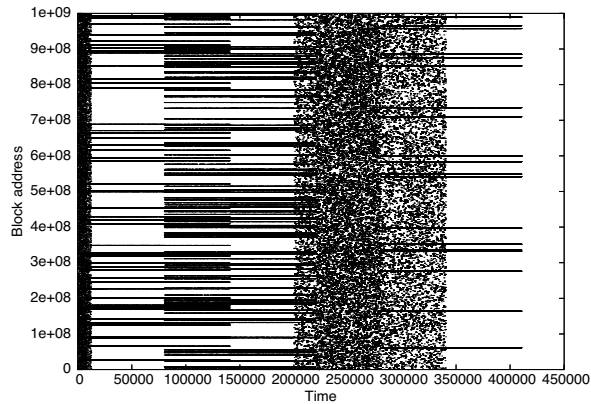


Figure 2: A visualization of the workload used for the TaP cache management simulation

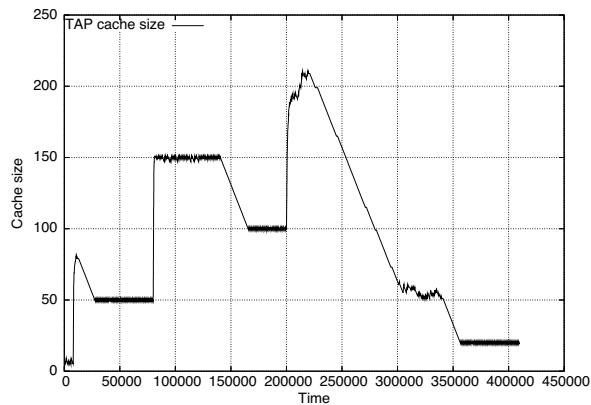


Figure 3: A sample of the behavior of the TaP cache manager

decrement module, by maintaining a downward pressure on the cache size, prevents cache size inflation. Together, the two modules force the cache size to ride close to a value optimal for the current workload.

4.2 A comparison of TaP and CaP

Figure 5 shows the results of a comparison of the TaP and CaP techniques on the basis of the cache size required by each to achieve the best possible hit rate on a given workload. In this experiment, we generated a synthetic workload of the following type. The workload is composed of a total of 50 streams, each of which arrives at an instant chosen uniformly at random in the simulation. Each stream in the workload is either completely sequential or completely random and is of a fixed finite duration much smaller than the length of the simulation. The number of completely sequential streams is varied from 1 to 50 (X axis of Figure 5), the remainder of the streams are random. First, the workload is run through a cache managed by the TaP technique with a table length of 1000 entries,

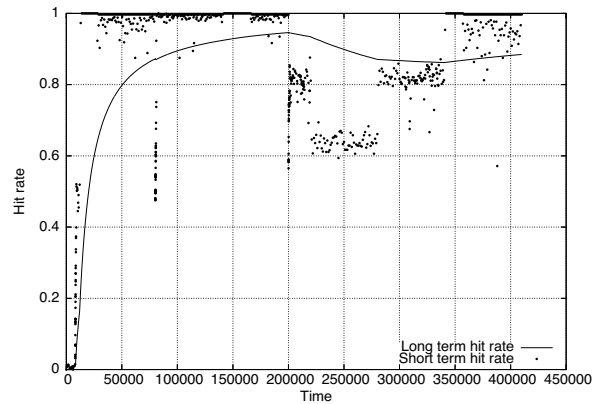


Figure 4: Short and long term hit rate obtained using the TaP cache manager

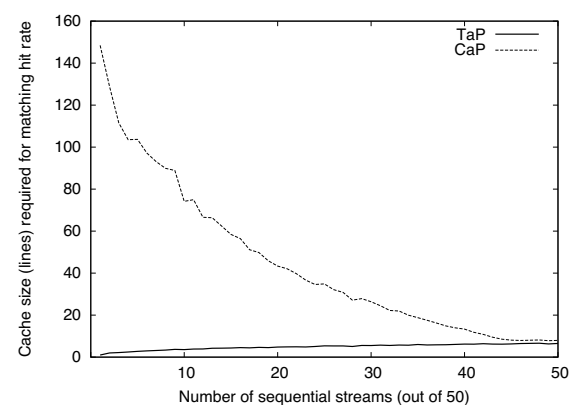


Figure 5: A comparison of cache sizes required by TaP and CaP to achieve matching hit rates as a function of the number of sequential streams (from 1 to 50).

and the hit rate achieved by TaP is recorded. Next, in order to find the cache size at which the CaP technique achieves the same hit rate as TaP, a series of experiments are conducted using the same workload as offered to TaP. We define the hit rate achieved by CaP to be the same as TaP when the relative difference between the two rates is no more than 5%.

Clearly, the cache size required by CaP to achieve the same hit rate as TaP is almost an order of magnitude larger when the number of sequential streams is small. This underscores the effectiveness of TaP's table-based techniques in two ways. First, it shows that TaP can detect and exploit sequentiality with a small prefetch cache size even when the sequentiality is latent and interleaved in a large amount of random data. This improvement in detection provides a significant reduction in the response time for individual sequential streams, even when the average hit rate of the workload is arbitrarily low. Thus, TaP succeeds in "connecting the dots" while using a minimal amount of cache resource. The additional

resource used by TaP, the table, is only a fraction (2-10 cache blocks) of the size of the cache. Second, as a result of the superior detection capability, the TaP cache manager's table-based cache inadequacy indicator can accurately and promptly respond to changes in a non-stationary workload.

4.3 SPC2-like workload

SPC2 [3, 18] is a popular benchmark that simulates workloads generated by applications that access their workload sequentially. Since we do not have access to the official SPC2 workload generator, we generated the workload using SPC2 published specifications [3]. The SPC2 workload is an interleaved mixture of highly sequential streams. Hence, all prefetched data would eventually result in hits if the data remain in the cache until their corresponding I/O requests arrive. In our experimental evaluation, we measure the mean hit rate and the response time as a function of some control variable such as the number of sequential streams or the cache size. However, in each experimental run, the cache size must be held constant for two reasons. First, the competing algorithms (AP, PoM, CaP) require a constant cache size. Second, we are interested in measuring performance given a fixed cache size. Therefore, TaP's dynamic cache sizing function is disabled for all the experiments in the sequel. We compare the performances of the various techniques under similar workloads and cache (+ table) sizes. A side-effect of turning off the prefetch cache sizing function is that the `replaceFlag` has no impact. That is, when a prefetch request is thrown out of the prefetch cache by the replacement scheme, its address is not put in the TaP table.

In our experiments, the number of streams is varied from 1 to 500. Depending on the cache size and prefetching technique, some of the prefetched data may get thrown out before they are used. PoM and NP perform far worse than the other techniques for obvious reasons, so below we analyze the results for TaP, CaP, and AP. From Figure 6 (a) one can see that when there is sufficient cache space to store at least one request from each of the streams, the cache hit rate is close to 1 for AP, TaP, and CaP (while PoM has a hit rate of 0.5 and NP has a hit rate of 0). As the number of sequential streams increases (beyond 40), the cache is no longer large enough to hold data from all the streams. Therefore, the hit rate gradually decreases.

We first compare the TaP with the CaP technique as the number of streams increases. TaP performs better than CaP since the small cache size makes it difficult for CaP to identify sequential streams. As a result, the total number of prefetches for CaP are far fewer than for TaP as shown in Figure 6 (c). We define the *useful prefetch*

ratio as the total number of prefetched requests that result in hits divided by the total number of prefetched requests. CaP stores random data for sequentiality detection thereby increasing the probability that prefetched data are thrown out before being used. Therefore, CaP has a lower useful prefetch ratio than TaP as shown in Figure 6 (d). Overall, TaP has a higher hit rate and a lower response time (Figure 6 (a) and (b)) than CaP.

We now compare the TaP and the AP techniques. TaP and AP perform similarly when the cache size is large enough to store data from all the streams. As the number of streams increases, the performance of AP is worse than that of TaP (Figure 6 (a) and (b)), and the reason can be seen by studying Figure 6 (c) and (d). As the number of streams increases relative to the cache size, AP continues to prefetch more than TaP. When a prefetched request is thrown out by the replacement scheme, TaP treats the next request from this stream as a random request and does not prefetch. (The address of the replaced request is not inserted into the TaP table since the cache size is not dynamically increased in this experimental evaluation.) AP prefetches on hits/misses while TaP only prefetches on hits. When the cache is too small for the workload, AP's useful prefetch ratio is much smaller than TaP's useful ratio, and as a result AP performs worse than TaP. The comparison between AP and TaP shows the negative impact of prefetching for this highly sequential workload when the cache size is too small to hold data from all the sequential streams. Figure 7 underscores this point: here, the cache size is very small. The performance of AP is comparable to CaP up to a certain point, but as the number of streams continues to increase, the performance of AP becomes worse than NP. Thus, for really small caches, never prefetch is better than always prefetch even when all the streams are highly sequential. TaP outperforms all the techniques evaluated for this cache size and workload. The mean response time for TaP is 20% lower than that of the other techniques.

4.4 Mix of 100% sequential streams and 100% random streams

In this set of experiments, a synthetic workload is used. The cache size and the total number of streams is fixed. Figure 8 shows the performance of the various techniques as the number of sequential streams is increased. The hit rate of all techniques (except NP) increases as the number of sequential streams increases. TaP consistently performs better than the other techniques and shows more improvement than the other techniques as the number of sequential streams increases. CaP performs worse than TaP, AP, and PoM as the number of sequential streams increases since CaP's sequential detection module is inefficient for this workload. AP and

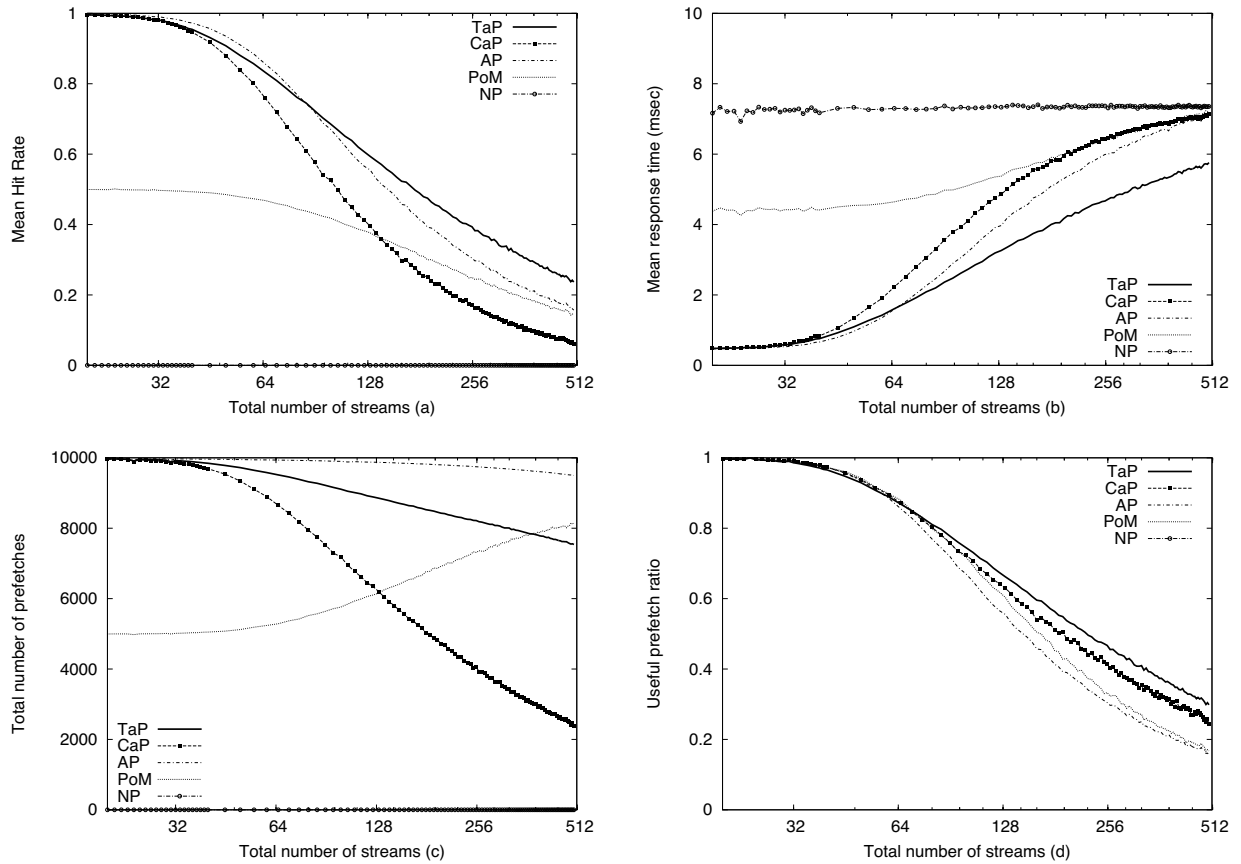


Figure 6: Performance of prefetching techniques with a cache of 4MB, table of 4KB, and the SPC2-like workload.

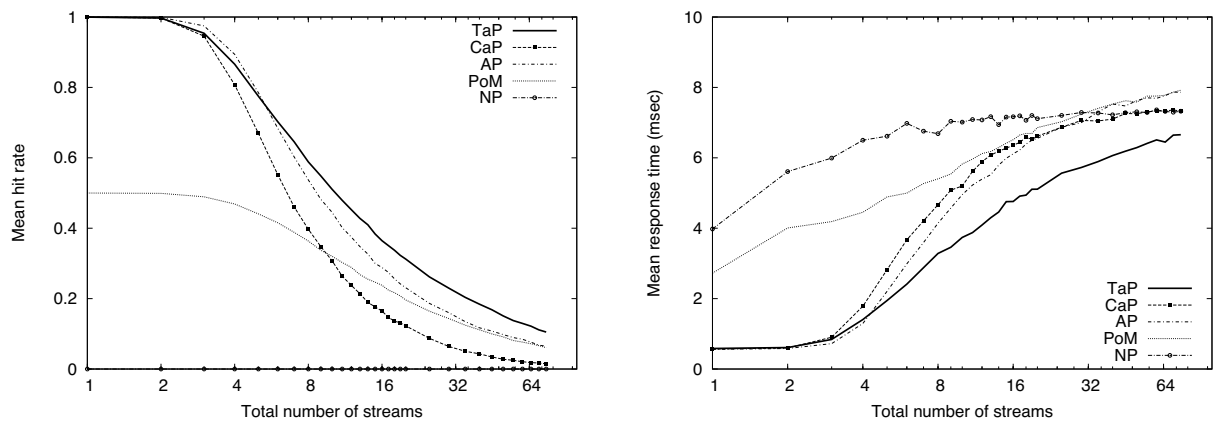


Figure 7: Performance of prefetching techniques with a cache of 240KB, table less than 0.8KB, and the SPC2-like workload.

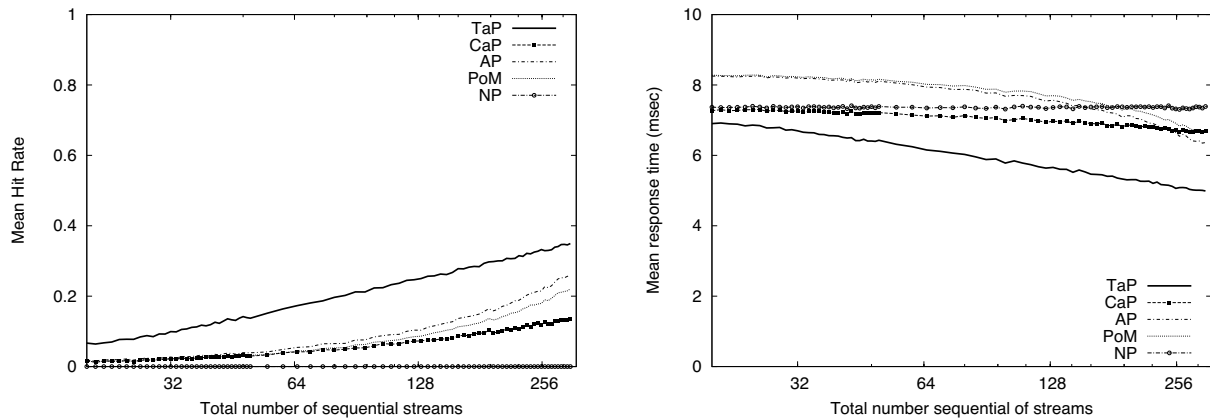


Figure 8: Performance of prefetching techniques with a cache of 4MB, table of 4KB, on a workload consisting of 300 streams of which m (X axis) are completely sequential and all others are completely random.

PoM have higher response time than NP when there are few sequential streams, but this changes as the fraction of sequential streams increases.

4.5 Varying cache sizes

This set of experiments demonstrates the efficient use of cache space by the TaP technique. In the first set of experiments (Figure 9), the workload is fixed at 20 random streams and 60 sequential streams. Regardless of the cache size, TaP consistently performs better than the other techniques because of TaP's efficient sequentiality detection module. In most cases, TaP uses less than half the cache space that CaP uses to get the same performance. For example, Figure 9 (b) shows that in order to keep response time under 4ms, TaP needs only 2MB while CaP and AP need more than 4MB. (PoM and NP are never able to achieve a response time of 4ms in this set of experiments.)

In the second set of experiments (Figure 10), the workload is fixed at 60 random streams and 20 sequential streams (*i.e.*, the number of sequential streams is decreased making the workload more random). With just 25% of the workload consisting of sequential streams, TaP is still able to detect the small degree of sequential streams with very small cache sizes. Both AP and CaP perform poorly for different reasons: AP's failure results from prefetching both random and sequential stream data into a small cache, so prefetched data from sequential streams get thrown out before they result in hits. CaP's failure results from its dependence on the cache size for sequentiality detection; the small cache fills quickly with random data, so sequential stream data are thrown out before they can be used for sequentiality detection. The efficient use of the cache by TaP is highlighted by this experiment. For example, when the prefetch cache is about

1MB, TaP's hit rate is about 4 times higher than all other techniques; and to achieve a response time of 6ms, TaP uses only 1 MB while CaP uses 4MB.

4.6 SPC1-like-read workload

SPC1 [2, 22, 30] is a popular benchmark that simulates workloads generated by business applications. Since we do not have access to the official SPC1 workload generator, we use a freely available alternative SPC1 workload generator [14]. We modified the workload by ignoring all write requests. Thus, the final workload is a SPC1-like-read workload. The number of Business Scaling Units (BSUs) roughly corresponds to the number of users generating the workload [14]. Therefore, the number of BSUs roughly corresponds to the number of workload streams.

We fix the cache size and then study the effect of increasing the number of BSUs. Figure 11 shows that the hit rate of the cache is small even with 1 BSU (hit rate < 0.3). This implies that the SPC1 workload has low degree of sequentiality. As the number of BSUs increase, the hit rate of all the prefetching strategies decreases. This implies that the cache is too small and that the degree of sequentiality per stream is low. Even for a workload with such low sequentiality, TaP gives the best hit rate and the lowest response time.

5 Conclusion

The TaP technique belongs to the class of Prefetch-on-Hit (PoH) techniques. Unlike existing PoH techniques that use the read cache to detect sequential streams in the I/O workload, the TaP technique uses a table to detect sequential streams. The use of a table by TaP ensures

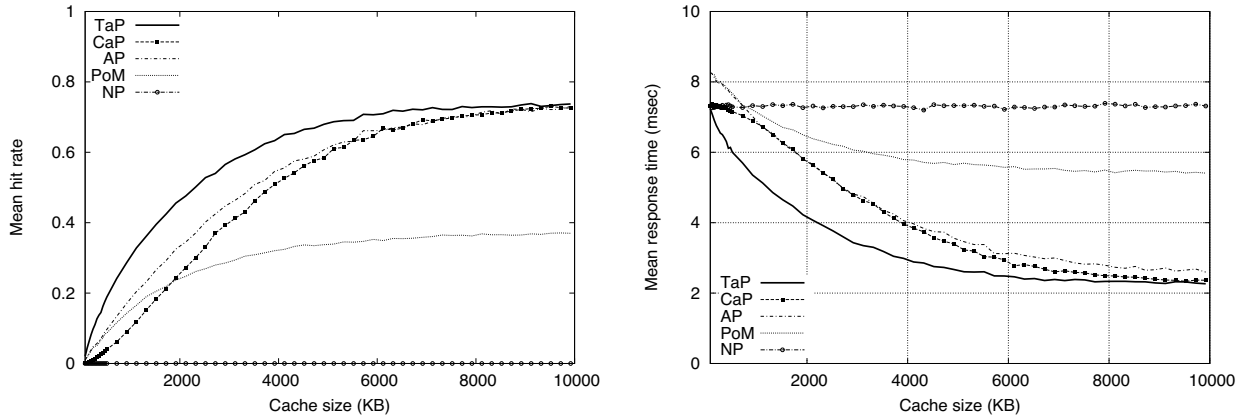


Figure 9: Impact of the cache size on the performance of prefetching techniques, total 80 streams, 60 completely sequential

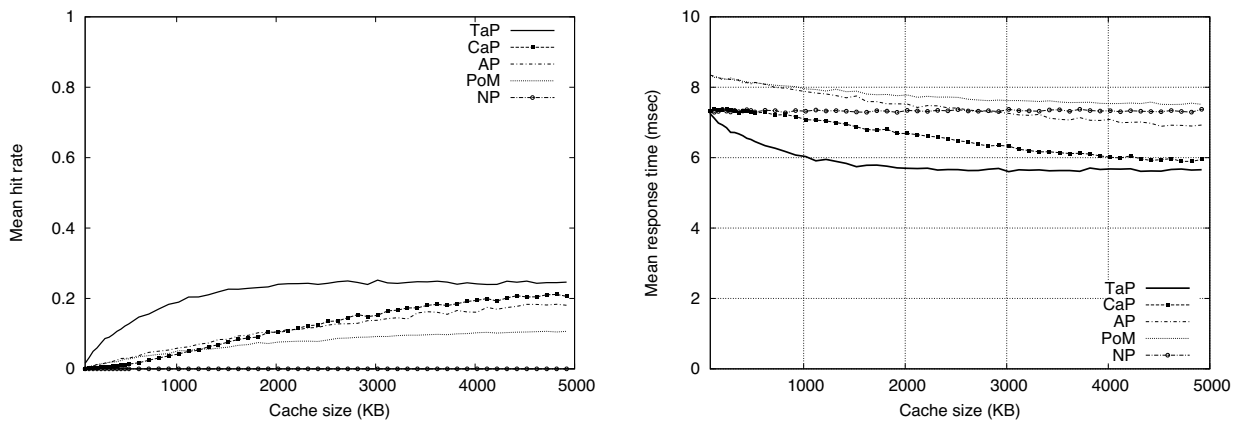


Figure 10: Impact of the cache size on the performance of prefetching techniques, total 80 streams, 20 completely sequential

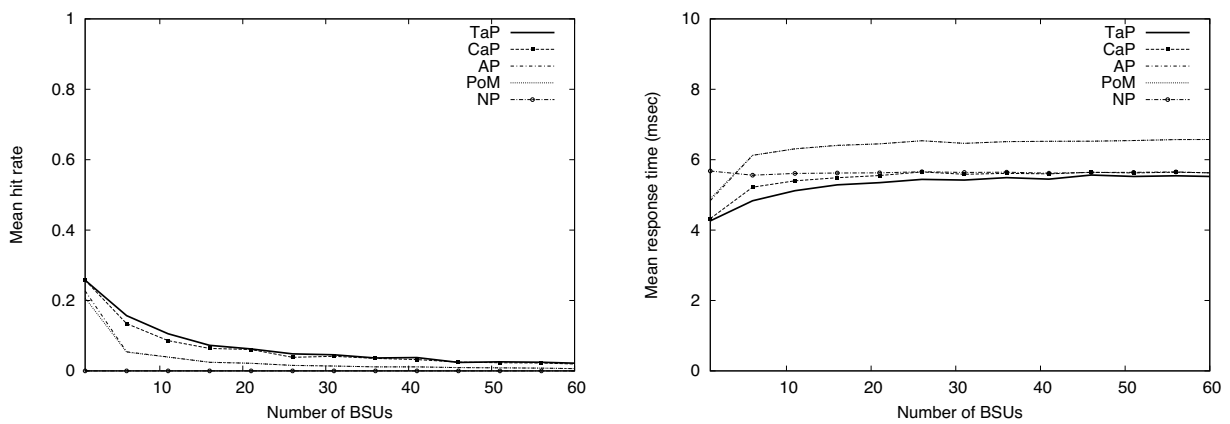


Figure 11: Performance of prefetching techniques as the number of BSUs (streams) in the SPC-1 workload is varied with cache size of 240KB

that a longer history of request access patterns can be tracked by the sequential detection module. This feature is very useful since I/O workloads consist of interleaved requests from various applications.

A unique feature of TaP is that the prefetch cache size is adjusted dynamically based on cache usage information from the TaP table. When the I/O workload has few sequential streams, the prefetch cache size is decreased and vice-versa. Our evaluation shows that for most workloads, TaP performs better than the other techniques for the smallest sized prefetch cache. TaP is superior to cache based PoH (CaP) techniques when the workload intensity is high and there is a mixture of sequential, partly sequential, and random workloads. At this point, the cache and disks are heavily utilized. The CaP technique wastes valuable cache space storing old data for sequential stream detection, particularly when the re-reference rate is low, as is often the case.

As future work, we plan to develop an integrated table-based technique that extracts both re-reference and sequential stream information from the I/O workload. Currently, re-reference data are detected when old I/O request data in the cache are hit. Prior studies have shown that most of the I/O workload is not re-referenced. However, a small fraction of the I/O workload gets many re-reference hits [33, 40]. The use of a table shows promise in detecting this small fraction of highly re-referenced data and managing the size of the re-reference cache.

Acknowledgments

We thank our anonymous reviewers for their helpful comments. We also thank Randal Burns, our shepherd, for directing the camera-ready version of this paper. This work was supported in part by the US National Science Foundation under CAREER grant CCR-0093111.

References

- [1] ZFS performance. online, 2007. http://www.solarisinternals.com/wiki/index.php/ZFS_Performance.
- [2] SPC Benchmark-1 (SPC-1) Official Specification, revision 1.10.1. Tech. rep., Effective 27 Sept. 2006. <http://www.storageperformance.org/specs>.
- [3] SPC Benchmark-2 (SPC-2) Official Specification, version 1.2.1. Tech. rep., Effective 27 Sept. 2006. <http://www.storageperformance.org/specs>.
- [4] ARI, I., AMER, A., GRAMACY, R., MILLER, E. L., BRANDT, S. A., AND LONG, D. D. E. ACME: adaptive caching using multiple experts. In *Proceedings of the 2002 Workshop on Distributed Data and Structures (WDAS)* (2002), Carleton Scientific. Extended version of the WDAS 2002 workshop paper.
- [5] BUCY, J. S., AND GANGER, G. R. The DiskSim simulation environment version 3.0 reference manual. Tech. Rep. CMU-CS-03-102, Carnegie Mellon University, School of Computer Science, January 2003.
- [6] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (1995), ACM Press, pp. 188–197.
- [7] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems* 14, 4 (1996), 311–343.
- [8] CHEN, T.-F., AND BAER, J.-L. Effective hardware based data prefetching for high-performance processors. *IEEE Trans. computers* 44, 5 (1995), 609–623.
- [9] CHI, C.-H., AND CHEUNG, C.-M. Hardware-driven prefetching for pointer data references. In *Proceedings of the 12th international conference on Supercomputing ICS* (1998), ACM.
- [10] CHRYSOS, G. Z., AND EMER, J. S. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium* (1998), Computer Architecture, 1998, pp. 142–153.
- [11] CORTES, T., AND LABARTA, J. Linear aggressive prefetching: A way to increase the performance of cooperative caches. In *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing* (1999), pp. 45–54.
- [12] CUREWITZ, K. M., KRISHNAN, P., AND VITTER, J. S. Practical prefetching via data compression. In *Proceedings of the ACM SIGMOD* (1993), International Conference on Management of Data archive, pp. 257 – 266.
- [13] DAHLGREN, F., DUBOIS, M., AND STENSTROM, P. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *International Conference on Parallel Processing* (1993), IEEE Computer Society, pp. 56–63.
- [14] DANIEL, S., AND FAITH, R. E. A portable, open-source implementation of the SPC-1 workload. In *Proceedings of the IEEE International Workload Characterization* (2005).
- [15] DOMENECH, J., SAHUQUILLO, J., GIL, J. A., AND PONT, A. The impact of the web prefetching architecture on the limits of reducing user's perceived latency. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence* (2006), IEEE Computer Society, pp. 740–744.
- [16] FARLEY, M. *Storage Networking Fundamentals: An Introduction to Storage Devices, Subsystems, Applications, Management, and Filing Systems*. Cisco Press, 2004.
- [17] FU, J. W. C., AND PATEL, J. H. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th annual international symposium on computer architecture* (1991), Computer Architecture, pp. 54–63.
- [18] GILL, B. S., AND BATHEN, L. A. D. AMP: Adaptive multi-stream prefetching in a shared cache. In *Proc. of USENIX 2007 Annual Technical Conference* (Feb 2007), 5th USENIX Conference on File and Storage Technologies.
- [19] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in adaptive replacement cache. In *Proc. of USENIX 2005 Annual Technical Conference* (2005), pp. 293–308.
- [20] GRIFFIOEN, J., AND APPLETON, R. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference* (1994), vol. 1, USENIX Association Berkeley, CA, USA, pp. 197–208.
- [21] GRIMSRUD, K. S., ARCHIBALD, J. K., AND NELSON, B. E. Multiple prefetch adaptive disk caching. *IEEE Trans. Knowl. Data Eng.* 5, 1 (1993), 88–103.

- [22] JOHNSON, S., MCNUTT, B., AND REITH, R. The making of a standard benchmark for open system storage. In *J. Comput. Resource Management* (Winter 2001), no. 101, pp. 26–32.
- [23] KALLAHALLA, M., AND VARMAN, P. J. Optimal prefetching and caching for parallel I/O systems. In *SPAA '01: Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2001), ACM Press, pp. 219–228.
- [24] KAREDLA, R., LOVE, J. S., AND WHERRY, B. G. Caching strategies to improve disk system performance. *Computer* 27, 3 (1994), 38–46.
- [25] KIMBREL, T., AND KARLIN, A. R. Near-optimal parallel prefetching and caching. In *SIAM Journal on Computing Archive* (2000), vol. 29, Society for Industrial and Applied Mathematics, pp. 1051 – 1082.
- [26] KIMBREL, T., TOMKINS, A., PATTERSON, R. H., CAO, B. B. P., FELTEN, E. W., GIBSON, G. A., KARLIN, A. R., AND LI, K. A trace-driven comparison of algorithms for parallel prefetching and caching. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI '96)* (1996), pp. 19–34.
- [27] LEE, R. L., YEW, P. C., AND LAWRIE, D. H. Data prefetching in shared memory multiprocessors. In *International Conference on Parallel Processing* (1987), IEEE Computer Society, pp. 28–31.
- [28] LEI, H., AND DUCHAMP, D. An analytical approach to file prefetching. In *Proceedings of the USENIX 1997 Annual Technical Conference* (1997), pp. 275–288.
- [29] LIANG, S., JIANG, S., AND ZHANG, X. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on* (2007), pp. 64–.
- [30] MCNUTT, B., AND JOHNSON, S. A standard test of I/O cache. In *Proceedings on Computer Measurement Group's 2001 International Conference* (2001).
- [31] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies (FAST '03)* (2003), pp. 115–130.
- [32] MOHAN, T., DE SUPINSKI, B. R., MCKEE, S. A., MUELLER, F., AND YOO, A. A quantitative measure of memory reference regularity. In *International Parallel and Distributed Processing Symposium* (April 2002).
- [33] MUNTZ, D., AND HONEYMAN, P. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter 1992 Technical Conference* (San Francisco, CA, USA, 1992), pp. 305–313.
- [34] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. SOSP Conf.* December, 1995.
- [35] PENDSE, R., AND BHAGAVATHULA, R. Pre-fetching with the segmented LRU algorithm. *Circuits and Systems, 1999. 42nd Midwest Symposium on* 2, 3 (1999), 862–865.
- [36] SMITH, A. J. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems* 3, 3 (1978), 223–247.
- [37] SMITH, A. J. Cache memories. *ACM Computing Surveys* 14, 3 (1982), 473–530.
- [38] TCHEUN, M. K., YOON, H., AND MAENG, S. R. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *International Conference on Parallel Processing* (1997), IEEE Computer Society, pp. 306 – 313.
- [39] VARKI, E., MERCHANT, A., XU, J., AND QIU, X. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 559–574.
- [40] WONG, T. M., AND WILKES, J. My cache or yours? making storage more exclusive. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), USENIX Association, pp. 161–175.
- [41] ZILLES, C., AND SOHI, G. Execution-based prediction using speculative slices. In *Proceedings of the 28th annual international symposium on Computer architecture* (2001), pp. 1–13.

The RAID-6 Liberation Codes

James S. Plank

*Department of Electrical Engineering and Computer Science
University of Tennessee*

Abstract

The RAID-6 specification calls for a storage system with multiple storage devices to tolerate the failure of any two devices. Numerous erasure coding techniques have been developed that can implement RAID-6; however, each has limitations. In this paper, we describe a new class of RAID-6 codes called the *Liberation Codes*. These codes encode, update and decode either optimally or close to optimally. Their modification overhead is lower than all other RAID-6 codes, and their encoding performance is often better as well. We provide an exact specification of the Liberation Codes and assess their performance in relation to other RAID-6 coding techniques. In the process, we describe an algorithm called *bit matrix scheduling*, which improves the performance of decoding drastically. Finally, we present a freely available library which facilitates the use of Liberation Codes in RAID-6 systems.

1 Introduction

As storage systems have grown in size and complexity, applications of RAID-6 fault-tolerance have become more pervasive. RAID-6 is a specification for storage systems composed of multiple storage devices to tolerate the failure of any two devices. In recent years, RAID-6 has become important when a failure of one disk drive occurs in tandem with the latent failure of a block on a second drive [9]. On a standard RAID-5 system, this combination of failures leads to permanent data loss. Hence, storage system designers have started turning to RAID-6.

Unlike RAID-1 through RAID-5, which detail exact techniques for storing and encoding data to survive single disk failures, RAID-6 is merely a specification. The exact technique for storage and encoding is up to the implementor. Various techniques for implementing RAID-6 have been developed and are based on *erasure codes* such as Reed-Solomon coding [2, 26], EVENODD cod-

ing [3] and RDP coding [9]. However, all of these techniques have limitations — there is no one *de facto* standard for RAID-6 coding.

This paper offers an alternative coding technique for implementing RAID-6. We term the technique *The RAID-6 Liberation Codes*, as they give storage systems builders a way to implement RAID-6 that frees them from problems of other implementation techniques. We give a complete description of how to encode, modify and decode RAID-6 systems using the Liberation Codes. We also detail their performance characteristics and compare them to existing codes.

The significance of the Liberation Codes is that they provide performance that is optimal, or nearly optimal in all phases of coding. They outperform all other RAID-6 codes in terms of modification overhead, and in many cases in encoding performance as well. We provide a freely available library that implements the various pieces of Liberation Coding. As such, we anticipate that they will become very popular with implementors of RAID-6 systems.

2 RAID-6 Specification and Current Implementations

RAID-6 is a specification for storage systems with $k + 2$ nodes to tolerate the failure of any two nodes. Logically, a typical RAID-6 system appears as depicted in Figure 1. There are $k + 2$ storage nodes, each of which holds B bytes, partitioned into k data nodes, D_0, \dots, D_{k-1} , and two coding nodes P and Q . The entire system can store kB bytes of data, which are stored in the data nodes. The remaining $2B$ bytes of the system reside in nodes P and Q and are calculated from the data bytes. The calculations are made so that if any two of the $k + 2$ nodes fail, the data may be recovered from the surviving nodes.

Actual implementations optimize this logical configu-



Figure 1: Logical overview of a RAID-6 system.

ration by setting B to be smaller than each disk's capacity, and then rotating the identity of the data and coding devices every B bytes. This helps remove hot spots in the system in a manner similar to RAID-5 systems. A pictorial example of this is in Figure 2. For simplicity, in the remainder of this paper we assume that each storage node contains exactly B bytes as in Figure 1 since the extrapolation to systems as in Figure 2 is straightforward.

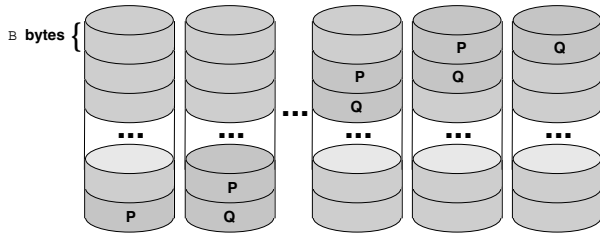


Figure 2: In actual implementations, the identities of the data and coding nodes rotate every B bytes. This helps to alleviate hot spots on the various drives.

The P device in RAID-6 is calculated to be the parity of the data devices. In this way, RAID-6 systems extrapolate naturally from RAID-5 systems by simply adding a Q drive. It also means that the sole challenge in designing a RAID-6 coding methodology lies in the definition of the Q drive. This definition must result in a *maximum distance separable (MDS)* code, which means that the Q drive cannot hold more than B bytes, and the original data must be restored following the failure of any two of the $k + 2$ devices.

There are several criteria that a storage system designer must evaluate when selecting an erasure coding technique for a RAID-6 system:

- *Encoding performance* is the speed of calculating P and Q from $D_0 \dots D_{k-1}$.
- *Modification performance* is the speed of recomputing relevant parts of P and Q when one of the D_i 's is modified.
- *Decoding performance* is the speed of recalculating lost data or coding information following one or two failures.
- *Ease of Implementation* is the complexity of the technique.

- *Cost of Implementation* pertains to licensing issues, as many erasure coding techniques are patented.

Below, we detail current techniques for implementing RAID-6.

Reed-Solomon Coding [26] is a very powerful general-purpose coding technique. It involves breaking up the data on each device into w -bit words, and then having the i -th word on the Q device be calculated from the i -th word on each data device using a special kind of arithmetic called *Galois Field* arithmetic ($GF(2^w)$). Galois Field addition is equivalent to the XOR operation; multiplication is much more difficult and requires one of a variety of techniques for implementation. As such, Reed-Solomon Coding is expensive compared to the other techniques. Reed-Solomon Coding is described in every text on coding theory [18, 19, 20, 27] and has tutorial instructions written explicitly for storage systems designers [21, 24].

Reed-Solomon Coding for RAID-6: Recently, Anvin has described a clever optimization to Reed-Solomon encoding for RAID-6 [2], based on the observation that multiplication by two may be implemented very quickly when w is a power of two. This optimization speeds up the performance of Reed-Solomon encoding. It does not apply to modification or decoding.

Parity Array coding applies a different methodology which is based solely on XOR operations. It works logically on groups of w bits from each data and coding device. The data bits of device D_i are labeled $d_{i,0}, \dots, d_{i,w-1}$, and the coding bits are p_0, \dots, p_{w-1} and q_0, \dots, q_{w-1} for the P and Q devices respectively. The p bits are calculated to be the parity of their respective data bits:

$$p_j = d_{0,j} \oplus d_{1,j} \oplus \dots \oplus d_{k-1,j}.$$

The q bits are defined to be the parity of some other collection of the data bits, and this definition is what differentiates one parity array code from another. A parity array system for $k = 5$ and $w = 4$ is depicted in Figure 3.

Obviously, to be efficient from an implementation standpoint, parity array codes do not work on single bits, but instead on w groups of bytes per RAID-6 block. In this way, we are not performing XORs on bits, but on machine words, which is very efficient. Thus the block size B defined above is restricted to be a multiple of w and the machine's word size.

Cauchy Reed-Solomon Coding is a technique that converts a standard Reed-Solomon code in $GF(2^w)$ to a parity array code which works on groups of w bits [6]. This has been shown to reduce the overhead of encoding and decoding [25], but not to the degree of the codes that we describe next.

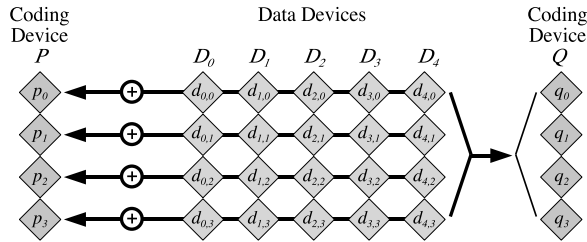


Figure 3: An example Parity Array code with $k = 5$ and $w = 4$. Logically, each element is a bit, but for efficient implementations, each element is a fixed-size group of words. Since there are w groups per device, the block size B for the code must be a multiple of w .

EVENODD coding departs from the realm of Reed-Solomon coding by defining the q_i bits from diagonal partitions of the data bits [3]. We do not provide an exact specification, but to give a flavor, we show the EVENODD code for $k = 5$ and $w = 4$ in Figure 4 (since the P device is parity, we do not picture it). The value S is an intermediate value used to calculate each q_i .

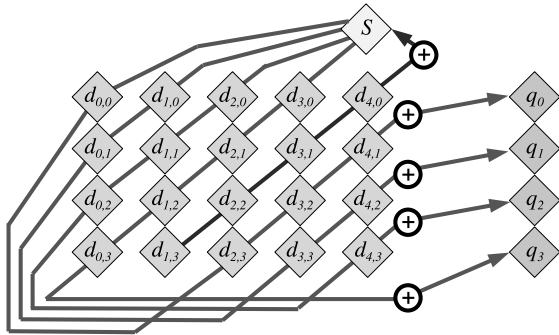


Figure 4: EVENODD coding with $k = 5$ and $w = 4$. The P device is not shown, as it is the parity of the data devices.

The parameter w must be selected such that $w + 1 \geq k$ and $w + 1$ is a prime number. Although this gives storage designers a variety of w to choose from for a given value of k , smaller w are more efficient than larger w .

EVENODD coding performs significantly better than all variants of Reed-Solomon coding. Its encoding performance is roughly $k - \frac{1}{2}$ XOR operations per coding word. Optimal encoding is equal to $k - 1$ XOR operations per coding word [4, 29]. Its modification performance is roughly three coding words per modified data word. Optimal is two. Finally, its decoding performance is roughly k XOR operations per failed word. As with encoding, optimal decoding performance is $k - 1$ XOR operations per failed word. Thus, EVENODD coding

achieves performance very close to optimal for both encoding and decoding. EVENODD coding was patented in 1996 [5].

RDP Coding is a parity array coding technique that is very similar to EVENODD coding, but improves upon it in several ways [9]. As with EVENODD coding, the number of bits per device, w , must be such that $w + 1$ is prime; however $w + 1$ must be strictly greater than k rather than $\geq k$. RDP calculates the bits of the Q device from both the data and parity bits, and in so doing achieves better performance. We show the RDP code for $k = 4$ and $w = 4$ in Figure 5.

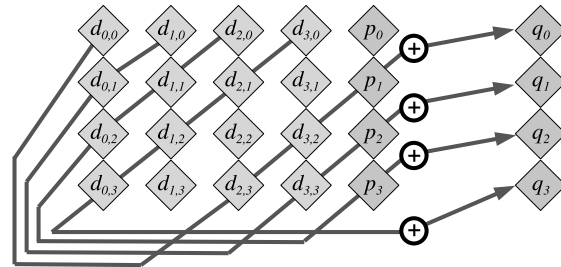


Figure 5: RDP coding with $k = 4$ and $w = 4$. As always, the p_j bits are the parity of the $d_{i,j}$ bits.

When $w = k$ or $w = k + 1$, RDP achieves optimal performance in both encoding and decoding. When $w \geq k + 2$, RDP still outperforms EVENODD coding and decoding, but it is not optimal. Like EVENODD coding, RDP coding modifies roughly three coding bits per modified data bit. RDP coding was patented in 2007 [10].

There are other very powerful erasure coding techniques that have been defined for storage systems. We do not address them in detail because they do not apply to RAID-6 systems as defined above. However, we mention them briefly. The X-Code [29] is an extremely elegant erasure code for two-disk systems that encodes, decodes and updates optimally. However, it is a *vertical* code that requires each device to hold two coding words for every k data words. It does not fit the RAID-6 specification of having coding devices P and Q , where P is a simple parity device.

The STAR code [17] and Feng's codes [11, 12] define encoding methodologies for more than two failures. Both boil down to EVENODD coding when applied to RAID-6 scenarios. There are other codes [13, 14, 15, 28] that tolerate multiple failures, but are not MDS, and hence cannot be used for RAID-6.

2.1 Why Do We Need Another Code?

Simply put, Reed-Solomon Coding is slow, and the parity array coding techniques exhibit suboptimal modifica-

tion performance and are patented. While patents should not have relevance to academic research papers, they do have a profound impact on those who implement storage systems and are the main reason why RAID-6 systems are still being implemented with Reed-Solomon coding. As such, alternative coding techniques that exhibit near-optimal performance are quite important.

Regardless of the patent issue, the Liberation codes have many properties that make them an attractive alternative to other RAID-6 techniques:

- They are parity array codes whose encoding performance is close to optimal. For all values of k , they outperform EVENODD encoding, and for some values of k , they even outperform RDP encoding. Thus, for many values of k , they represent the best known RAID-6 codes.
- To build flexible RAID-6 systems, it is often advantageous to allow k to grow and shrink dynamically within limits. For the parity array codes (including Liberation codes), this means employing a fixed value of w in all cases that can accommodate the largest possible value of k . EVENODD and RDP coding systems will work in this way, but their performance suffers when k shrinks, because they cannot compensate by decreasing w as well. In contrast, Liberation codes improve as w grows, and thus exhibit better performance in systems where k varies beneath a threshold value.
- Their modification performance is very close to the optimal value of two updated coding bits per modified data bit. This is an improvement on the other coding techniques, and it can be shown that it achieves the lower bound for all RAID-6 codes.
- The decoding performance is within 15% of optimal.
- Their implementation is freely available.

We describe the codes and analyze their performance below.

3 Liberation Code Description

Liberation coding and decoding are based on a bit matrix-vector product very similar to the those used in Reed-Solomon coding [18, 20] and Cauchy Reed-Solomon coding [6]. This product precisely defines how encoding and modification are performed. Decoding is more complex and to proceed efficiently, we must augment the bit matrix-vector product with the notion of “bit matrix scheduling.” We first describe the general methodology of bit matrix coding and then

define the Liberation Codes and discuss their encoding/modification performance. We then describe decoding, and how its performance may be improved by bit matrix scheduling. We compare the Liberation Codes to the other RAID-6 codes in Section 4.

3.1 Bit Matrix Coding Overview

Bit matrix coding is a parity array coding technique first employed in Cauchy Reed-Solomon coding [6]. In general, there are k data devices and m coding devices, each of which holds exactly w bits. The system uses a $w(k+m) \times wk$ matrix over $GF(2)$ to perform encoding. This means that every element of the matrix is either zero or one, and arithmetic is equivalent to arithmetic modulo two. The matrix is called a *binary distribution matrix*, or BDM. The state of a bit matrix coding system is described by the matrix-vector product depicted in Figure 6.

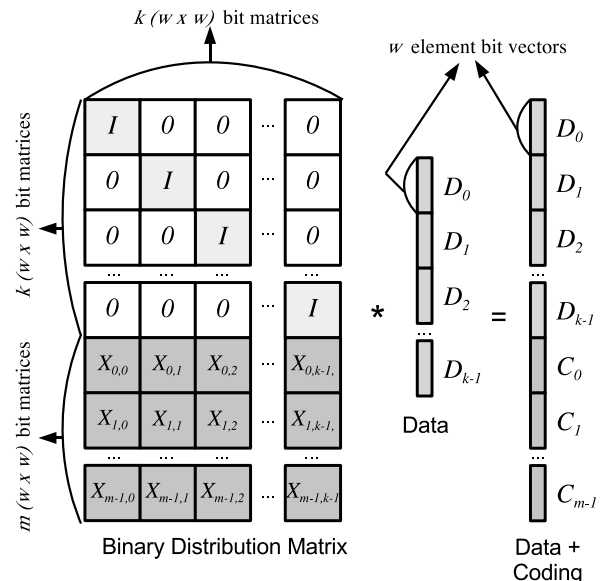


Figure 6: An example bit matrix coding system.

The BDM has a specific format. Its first wk rows compose a $wk \times wk$ identity matrix, pictured in Figure 6 as a $k \times k$ matrix whose elements are each $w \times w$ bit matrices. The next mw rows are composed of mk matrices, each of which is a $w \times w$ bit matrix $X_{i,j}$.

We multiply the BDM by a vector composed of the wk bits of data. We depict that in Figure 6 as k bit vectors with w elements each. The product vector contains the $(k+m)w$ bits of the entire system. The first wk elements are equal to the data vector, and the last mw elements contain the coding bits, held in the m coding devices.

Note that each device corresponds to a row of $w \times w$ matrices in the BDM, and that each bit of each device corresponds to one of the $w(k + m)$ rows of the BDM. The act of encoding is to calculate each bit of each C_i as the dot product of that bit's row in the BDM and the data. Since each element of the system is a bit, this dot product may be calculated as the XOR of each data bit that has a one in the coding bit's row. Therefore, the performance of encoding is directly related to the number of ones in the BDM.

To decode, suppose some of the devices fail. As long as there are k surviving devices, we decode by creating a new $wk \times wk$ matrix BDM' from the wk rows corresponding to k of the surviving devices. The product of that matrix and the original data is equal to these k surviving devices. To decode, we therefore invert BDM' and multiply it by the survivors – that allows us to calculate any lost data. Once we have the data, we may use the original BDM to calculate any lost coding devices.

For a coding system to be MDS, it must tolerate the loss of any m devices. Therefore, every possible BDM' matrix must be invertible. This is done in Cauchy Reed-Solomon coding by creating each $X_{i,j}$ from a Cauchy matrix in $GF(2^w)$ [6]. However, these do not perform optimally. It is an open question how to create optimally performing bit matrices in general.

Since the first wk rows of the BDM compose an identity matrix, we may precisely specify a BDM with a *Coding Distribution Matrix* (CDM) composed of the last wm rows of the BDM. It is these rows that define how the coding devices are calculated. (In coding theory, the CDM composes the leftmost wk columns of the parity check matrix).

3.2 RAID-6 Bit Matrix Encoding

When this methodology is applied to RAID-6, the BDM is much more restricted. First, $m = 2$, and the two coding devices are named $P = C_0$ and $Q = C_1$. Since the P device must be the parity of the data devices, each matrix $X_{0,i}$ is equal to a $w \times w$ identity matrix. Thus, the only degree of freedom is in the definition of the $X_{1,i}$ matrices that encode the Q device. For simplicity of notation, we remove the first subscript and call these matrices X_0, \dots, X_{k-1} . A RAID-6 system is depicted in Figure 7 for $k = 4$ and $w = 4$.

To calculate the contents of a coding bit, we simply look at the bit's row of the CDM and calculate the XOR of each data bit that has a one in its corresponding column. For example, in Figure 7, it is easy to see that $p_0 = d_{0,0} \oplus d_{1,0} \oplus d_{2,0} \oplus d_{3,0}$.

When a data bit is modified, we observe that each data bit $d_{i,j}$ corresponds to column $wi + j$ in the CDM. Each coding bit whose row contains a one in that column must

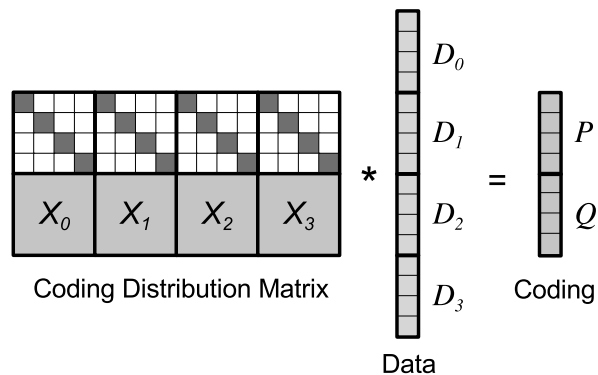


Figure 7: Bit matrix representation of RAID-6 coding when $k = 4$ and $w = 4$.

be updated with the XOR of the data bit's old and new values.

Therefore, to employ bit matrices for RAID-6, we are faced with a challenge to define the X_i matrices so that they have a minimal number of ones, yet remain MDS. A small number of ones is important for fast encoding and updating. We shall see the impact on decoding later in the paper.

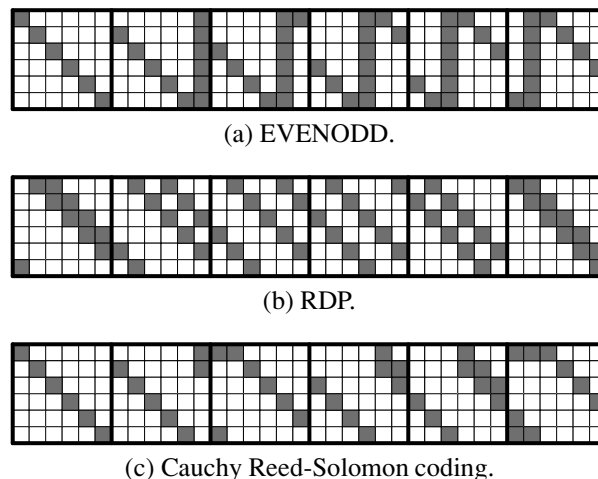


Figure 8: The X_i matrices defining the BDM's for various RAID-6 coding techniques, $k = 6$ and $w = 6$.

It is an interesting aside that *any* RAID-6 code based on XOR operations may be defined with a bit matrix. To demonstrate, we include the X_i for EVENODD, RDP and Cauchy Reed-Solomon coding when $k = 6$ and $w = 6$ in Figure 8. It is a simple matter to verify that each of these defines an MDS code.

There are 61 ones in the EVENODD matrices, 60 in the RDP matrices and 46 in the Cauchy Reed-Solomon matrices. Thus, were one to encode with the bit matri-

ces, the Cauchy Reed-Solomon coding matrices would be the fastest, which would seem to contradict the fact that RDP encodes optimally. We explore this more fully in Section 3.4 below, where we demonstrate how to improve the performance of bit matrix encoding so that it does not rely solely on the number of ones in the matrix.

The performance of updating, however, is directly related to the number of ones in each column, and there is no way to optimize that further. The fact that EVEN-ODD and RDP coding update must update roughly three coding bits per data bit is reflected in their CDM's, which have an average of $\frac{36+61}{36} = 2.69$ and $\frac{36+60}{36} = 2.67$ ones per column respectively (we add 36 ones for the identity matrices that encode the P device). The Cauchy Reed-Solomon CDM requires only 2.31 modifications per data bit.

3.3 Liberation Code Specification

We now define the Liberation codes. As with EVEN-ODD and RDP coding, the value of w is restricted and depends on k . In particular, w must be a prime number $\geq k$ and > 2 . To specify the X_i matrices, we use two pieces of notation:

- We define $I_{\rightarrow j}^w$ to be the $w \times w$ identity matrix whose columns have been rotated to the right by j columns. Note that $I^w = I_{\rightarrow 0}^w$.
- We define $O_{i,j}^w$ to be a $w \times w$ matrix where every element is zero, except for the element in row $(i \bmod w)$ and column $(j \bmod w)$, which equals one.

The Liberation codes are defined as follows:

- $X_0 = I^w$.
- For $0 < i < k$, $X_i = I_{\rightarrow i}^w + O_{y,y+i-1}^w$, where $y = \frac{i(w-1)}{2}$. An alternate and equivalent specification is that $y = \frac{w-i}{2}$ when i is odd, and $y = w - \frac{i}{2}$ when i is even.

Figure 9 shows the X_i matrices for the Liberation Code when $k = 7$ and $w = 7$. It may be proven that for all prime $w > 2$, the Liberation Code for $k \leq w$ is an MDS code. The complete proof is beyond the scope of this paper, and is instead in an accompanying technical report [23]. We provide a sketch of the proof at the end of this paper in Section 7.

For any given values of k and w , the X_i matrices have a total of $k w + k - 1$ ones. Add this to the $k w$ ones for device P 's matrices, and that makes $2 k w + k - 1$ ones in the CDM. If a coding bit's row of the CDM has o ones, it takes $(o - 1)$ XORs to encode that bit from the data bits. Therefore, each coding bit requires an average of $\frac{2 k w + k - 1 - 2 w}{2 w} = k - 1 + \frac{k - 1}{2 w}$ XORs. Optimal is $k - 1$.

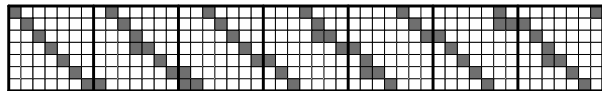


Figure 9: The X_i matrices for the Liberation Code when $k = 7$ and $w = 7$.

The average ones per column of the CDM is $\frac{2 k w + k - 1}{k w} = 2 + \frac{k - 1}{k w}$, which is roughly two. Optimal is two. Thus, the Liberation codes achieve near optimal performance for both encoding and modification. We explore the notion of optimality in terms of the number of ones in an MDS RAID-6 CDM in Section 6 below. There we will show that the Liberation Codes achieve the lower bound on number of ones in a matrix.

3.4 Bit Matrix Scheduling for Decoding

To motivate the need for bit matrix scheduling, consider an example when $k = 5$ and $w = 5$. We encode using the Liberation code, and devices D_0 and D_1 fail. To decode, we create BDM' by deleting the top 10 rows of the BDM and inverting it. The first 10 rows of this inverted matrix allow us to recalculate D_0 and D_1 from the surviving devices. This is depicted in Figure 10.

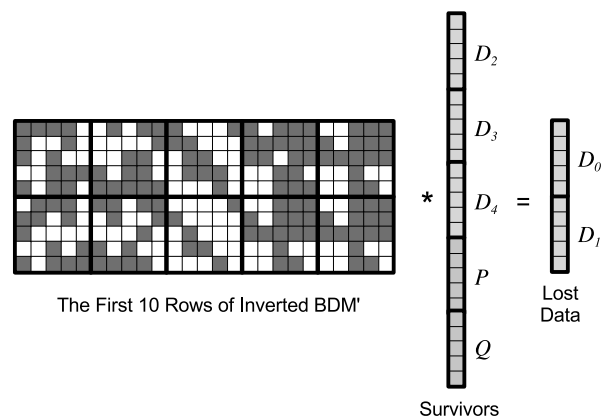


Figure 10: Decoding D_0 and D_1 from the Liberation Codes when $k = 5$ and $w = 5$.

Calculating the ten dot products in the straightforward way takes 124 XORs, since there are 134 ones in the matrix. Optimal decoding would take 40 XORs. Now, consider rows 0 and 5 of the matrix which are used to calculate $d_{0,0}$ and $d_{1,0}$ respectively. Row 0 has 16 ones, and row 5 has 14 ones, which means that $d_{0,0}$ and $d_{1,0}$ may be calculated with 28 XORs in the straightforward manner. However, there are 13 columns in which both rows have ones. Therefore, suppose we first calculate $d_{1,0}$,

which takes 13 XORs, and then calculate $d_{0,0}$ using the equation:

$$d_{0,0} = d_{1,0} \oplus d_{2,0} \oplus d_{3,0} \oplus d_{4,0} \oplus p_0.$$

This only takes four additional XOR operations, lowering the total for the two bits from 28 XORs to 17.

This observation leads us to a simple algorithm which we call **bit matrix scheduling**, for performing a collection of dot products in a bit matrix-vector product more efficiently than simply performing each dot product independently. To describe the algorithm, we use the following assumptions and notation:

- We are multiplying matrix M by vector V to calculate the product vector U . All elements are bits and arithmetic is in $GF(2)$.
- Matrix M has r rows and c columns. The element in row i , column j is denoted $M[i, j]$. Vector V has c elements denoted $V[0], \dots, V[c-1]$, and vector p has r elements denoted $U[0], \dots, U[r-1]$.
- We denote row i of M as M_i .
- $From[]$ is a vector of r integers, each initialized to -1.
- $Ones[]$ is a vector of r integers, initialized so that $Ones[i]$ equals the number of ones in row i of the matrix.
- $Sum(i, j)$ is a c -element bit vector that equals the sum (in $GF(2)$) of M_i and M_j .
- $Notdone$ is a set of integers initialized to contain all values in $[0..r-1]$.

The algorithm proceeds in r steps. Each step performs the following operations:

1. Select i such that $i \in Notdone$ and $Ones[i]$ is minimized. Break ties arbitrarily.
2. If $From[i]$ equals -1, then $U[i]$ is calculated to be the XOR of all $V[j]$ such that $M[i, j] = 1$. If $From[j]$ does not equal -1, then $U[i]$ is calculated as the XOR of $U[From[i]]$ and all $V[j]$ such that $M[i, j] + M[From[i], j] = 1$.
3. Remove i from $Notdone$.
4. For all $j \in Notdone$, calculate x to be one plus the number of ones in $Sum(i, j)$. If $x < Ones[j]$, then set $Ones[j]$ to x , and $From[j]$ to i .

Thus, if it is more efficient to calculate the product element from another product element than from the original vector, this algorithm makes that happen. When the algorithm operates on the example in Figure 10, it ends up with the following schedule:

- Calculate $d_{1,3}$: 7 XORs.
- Calculate $d_{0,3}$ from $d_{1,3}$: 4 XORs.
- Calculate $d_{1,4}$ from $d_{0,3}$: 5 XORs.
- Calculate $d_{0,4}$ from $d_{1,4}$: 4 XORs.
- Calculate $d_{1,0}$ from $d_{0,4}$: 5 XORs.
- Calculate $d_{0,0}$ from $d_{1,1}$: 4 XORs.
- Calculate $d_{1,1}$ from $d_{0,0}$: 4 XORs.
- Calculate $d_{0,1}$ from $d_{1,1}$: 4 XORs.
- Calculate $d_{1,2}$ from $d_{0,1}$: 5 XORs.
- Calculate $d_{0,2}$ from $d_{1,2}$: 4 XORs.

This is a total of 46 XORs, as opposed to 124 without scheduling. An optimal algorithm would decode with 40 XORs.

We note that this algorithm does not always yield an optimal schedule of operations. For example, one can encode using the matrix in Figure 8(a) (EVENODD coding with $k = 6$, $w = 6$) with exactly 41 XOR operations by first calculating $S = d_{1,5} \oplus d_{2,4} \oplus d_{3,3} \oplus d_{4,2} \oplus d_{5,1}$ and using S in each dot product. When the bit scheduling algorithm is applied to that matrix, however, it is unable to discover this optimization, and in fact yields no improvements in encoding: the dot products take 55 XORs.

However, for decoding with the Liberation Codes, this algorithm improves performance greatly. As an interesting aside, the algorithm derives the optimal schedule for both encoding and decoding using the bit matrix versions of RDP codes, and it improves the performance of both encoding and decoding with Cauchy Reed-Solomon coding. It is an open question to come up with an efficient algorithm that produces optimal schedules for all bit matrix-vector products.

3.5 Caching Schedules

The algorithm for bit matrix scheduling, like the inversion of the BDM' matrix, is $O(w^3)$. Since w is likely to be relatively small in a RAID-6 system, and since encoding and decoding both involve XORs of $O(w^2)$ distinct elements, the inversion and bit scheduling should not add much time to performance of either operation. However, since the total possible number of schedules is bounded by $\binom{k+2}{2}$, it is completely plausible to precalculate each of the $\binom{k+2}{2}$ schedules and cache them for faster encoding and decoding.

4 Performance

We have implemented encoding, modification and decoding using all the techniques described in this paper. In all the graphs below, the numbers were generated by instrumenting the implementation and counting the XOR operations. When there is a closed-form expression for a

metric (e.g., encoding with RDP, EVENODD, or Liberation codes), we corroborated our numbers with the expression to make sure that they matched.

4.1 Performance of Encoding

We measure the performance of encoding as the average number of XOR operations required per coding word. This includes encoding both the P and Q devices. Since optimal encoding is $k - 1$ XORs per coding word, we can normalize by dividing the number of XORs per coding word by $k - 1$ to achieve the overhead of the code: the factor of encoding performance worse than optimal performance. Thus, low values are desirable, the optimal value being one. These values are presented in Figure 11.

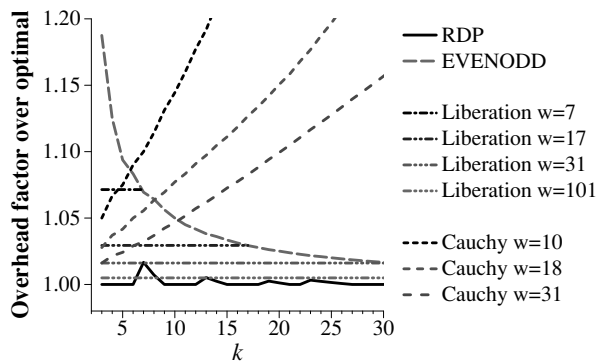


Figure 11: Encoding performance of various XOR-based RAID-6 techniques. Optimal encoding is $k - 1$ XORs per coding word.

RDP encoding achieves optimality when $k + 1$ and $k + 2$ are prime numbers. Otherwise, the code is *shortened* by assuming that there are data devices that hold nothing but zeros [9]. As the code is asymmetric, the best performance is achieved by assuming that the first $w - k$ devices are zero devices. This is as opposed to EVENODD coding, which performs best when the last $w - k$ devices are zero devices. With both RDP and EVENODD coding, w is a function of k , as smaller w perform better than larger w .

With Liberation codes, this is not the case – larger w perform better than smaller w . For that reason, we plot four values of w in Figure 11. The lines are flat, because the number of XORs per coding word (from Section 3.3) is equal to $k - 1 + \frac{k-1}{2w}$, and therefore their factor over optimal is $1 + \frac{1}{2w}$. As such, the codes are asymptotically optimal as $w \rightarrow \infty$. As a practical matter though, smaller w require the coding engine to store fewer blocks of data in memory, and may perform better than larger w due to memory and caching effects. The selection of a good w in Liberation Coding thus involves a tradeoff be-

tween the fewer XORs required by large w and the reduced memory consumption of small w .

The performance of EVENODD encoding is roughly $k - \frac{1}{2}$, which is worse than both RDP and Liberation encoding except when $k = w$ in Liberation Coding and the two perform equally.

The Cauchy Reed-Solomon codes for various w are also included in the graph. Like Liberation Codes, Cauchy Reed-Solomon codes perform better with larger w than with smaller w . However, unlike the other codes, their performance relative to optimal worsens as k grows. It is interesting to note that they outperform EVENODD coding for small k . Since their performance is so much worse than the others, we omit them in subsequent graphs.

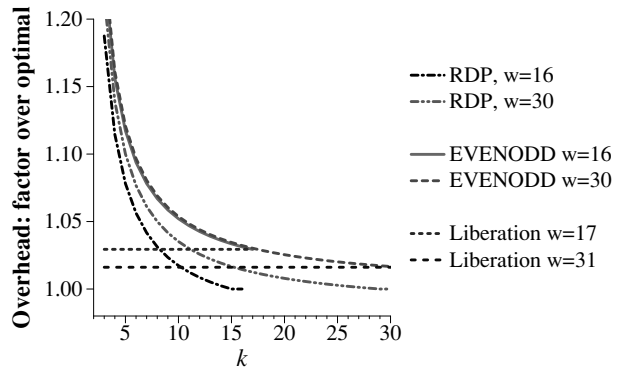


Figure 12: Encoding performance of RDP, EVENODD and Liberation codes when w is fixed.

One of the attractive features of these XOR codes is that if w is chosen to be large enough, then the same code can support any $k \leq w$ devices. Adding or subtracting devices only involves modification to coding devices P and Q , and does not require re-encoding the entire system as, for example, the X-Code would [29]. For that reason, Figure 12 shows the performance of RDP, EVENODD and Liberation encoding when w is fixed. Since RDP and EVENODD coding require $w + 1$ to be prime, and Liberation coding requires w to be prime, we cannot compare the same values of w , but values that are similar and that can support nearly the same number of data devices. Although RDP outperforms the Liberation codes for larger k , for smaller k , the Liberation codes perform better. Moreover, their performance relative to optimal is fixed for all k , which may ease the act of scheduling coding operations in a distributed storage system.

4.2 Performance of Modification

Figure 13 shows the average number of coding bits that must be modified when a data bit is updated. With both EVENODD and RDP coding, this number increases with w , reaching a limit of three as w grows. With Liberation codes, the opposite is true, as the number of modified coding bits is roughly two. Clearly, the Liberation codes outperform the other two in modification performance.

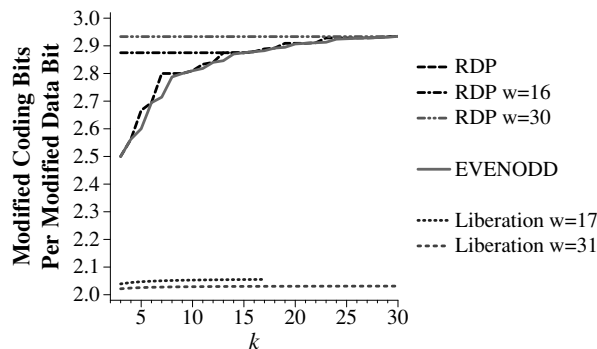


Figure 13: Modification performance of RDP, EVENODD and Liberation codes.

4.3 Performance of Decoding

For single failures, all RAID-6 systems decode identically. If the failure is in a data device, then it may be decoded optimally from the P device. Otherwise, decoding is identical to encoding. Thus, we only concern ourselves with two-device failures. To test decoding, we measured the performance of decoding for each of the $\binom{k+2}{2}$ possible combinations of failures. As with encoding, we measure number of XORs per failed word and present the average value. In Figure 14 we plot the measurements, again as a factor over optimal, which is $k - 1$ XORs per failed word.

In general, RDP coding exhibits the best decoding performance, followed by EVENODD coding and then Liberation coding, which decodes at a rate between ten and fifteen percent over optimal. The effectiveness of bit matrix scheduling is displayed in Figure 15, which shows the performance of Liberation decoding without scheduling for $w = 17$ and $w = 31$.

Figure 15 clearly shows that without bit scheduling, Liberation codes would be unusable as a RAID-6 technique. It remains a topic of future work to see if the scheduling algorithm of Section 3.4 may be improved further.

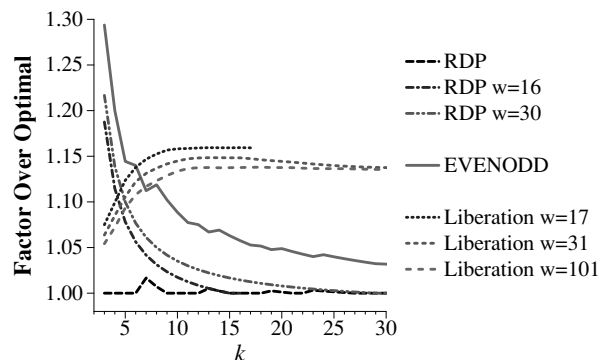


Figure 14: Decoding performance of RDP, EVENODD and Liberation codes.

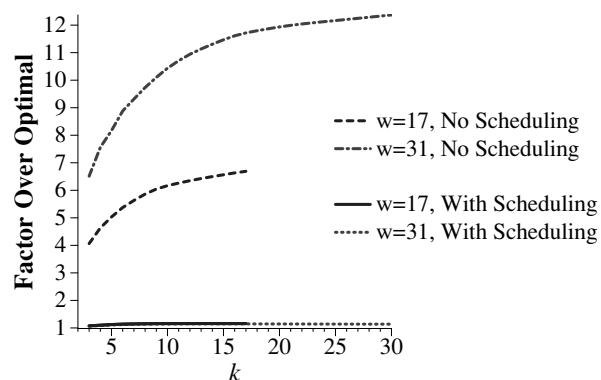


Figure 15: The effectiveness of bit matrix scheduling on Liberation decoding.

4.4 Comparison to Reed-Solomon Coding

We do not include a detailed comparison of Liberation Coding to standard Reed-Solomon coding. Instead, in Figure 16 we present measurements of the basic operations of Reed-Solomon coding on three different machines. The first machine is a MacBook Pro with a 2.16 GHz Intel Core 2 Duo processor. The second is a Dell Precision with a 1.5 GHz Intel Pentium processor. The third is a Toshiba Tecra with a 1.73 GHz Intel Pentium processor. On each, we measure the bandwidth of three operations: XOR, multiplication by an arbitrary constant in $GF(2^8)$ and multiplication by two using Anvin's optimization [2]. All operations are implemented using the **jerasure** library presented in section 5. In particular, multiplication by an arbitrary constant is implemented using a 256×256 multiplication table.

We may project the performance of standard and optimized Reed-Solomon coding as follows. Let B_{\oplus} be the bandwidth of XOR (in GB/s), B_{\otimes} be the bandwidth of arbitrary multiplication, and $B_{\otimes 2}$ be the bandwidth of multiplication by two. The time to encode one gigabyte

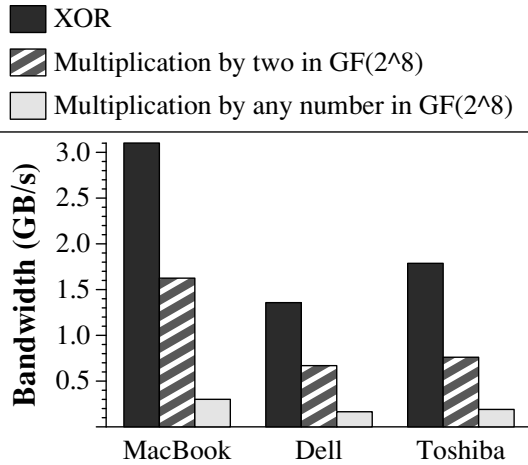


Figure 16: The bandwidth of basic operations for Reed-Solomon Coding.

of data on k devices using standard Reed-Solomon coding is:

$$2 * \frac{k-1}{B_{\oplus}} + \frac{k-1}{B_{\otimes}}.$$

This is because the P device is still encoded with parity, and the Q device requires $(k-1)$ multiplications by a constant. The time to encode one megabyte with Anvin's optimization simply substitutes B_{\otimes} with $B_{\otimes 2}$. Finally, optimal encoding time is $2 * \frac{k-1}{B_{\oplus}}$, reflecting $k-1$ XORs per coding word.

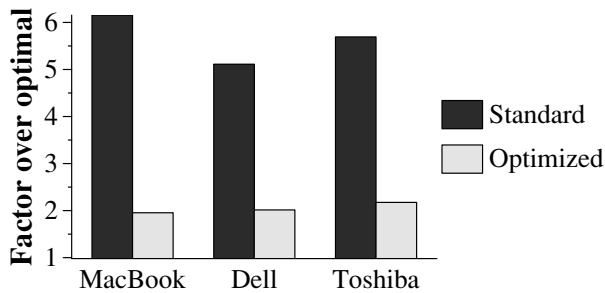


Figure 17: The projected performance of standard and optimized Reed-Solomon coding using the bandwidth measurements from Figure 16.

In Figure 17, we plot the performance of encoding using the bandwidth numbers from Figure 16. For standard Reed-Solomon coding, the performance of decoding is roughly equal to encoding performance. Anvin's optimization improves the performance of encoding roughly by a factor of three. However, it is much worse than the XOR-based codes. Moreover, the optimization does not apply to decoding, which will perform at the same rate as

standard Reed-Solomon coding. Thus, we conclude that even with the optimization, Reed-Solomon coding is an unattractive alternative for RAID-6 applications.

5 Liberation Coding Library

We have implemented a library in C/C++ to facilitate all Liberation coding operations. It is part of the **jerasure** library [22], which implements all manners of matrix and bit matrix coding, including regular Reed-Solomon coding, Cauchy Reed-Solomon coding and Liberation coding. The library is roughly 6000 lines of code and is freely available under the GNU LPL.

Table 1 lists some of the relevant procedures from the library. In all the procedures, \mathbf{k} , \mathbf{w} and \mathbf{B} are as defined in this paper, \mathbf{m} is the number of coding devices ($\mathbf{m}=2$ for RAID-6), \mathbf{data} and \mathbf{coding} are pointers to data and coding regions, and $\mathbf{totalsize}$ is the total number of bytes in each device. Note that $\mathbf{totalsize}$ must be a multiple of \mathbf{B} and the machine's word size. Bit matrices are represented as linear arrays of integers whose values are either zero or one. The element in row i and column j is in array element $ikw + j$.

The first procedure creates a schedule from a bit matrix, which may be an encoding or decoding bit matrix. The schedule is an array of operations, where each operation is itself an array of five integers:

$$\langle copy|xor, from_{id}, from_{bit}, to_{id}, to_{bit} \rangle,$$

where $copy|xor$ specifies whether the operation is to copy data or XOR it, $from_{id}$ is the id of the source device, $from_{bit}$ is the source bit number (i.e. a number from 0 to $w-1$), to_{id} is the id of the destination device and to_{bit} is the destination bit number. **jerasure_generate_schedule_cache()** creates a cache of all possible decoding schedules.

The two encoding routines encode using either a bit matrix or a schedule, and the three decoding routines decode using a bit matrix with no schedule, a bit matrix generating a schedule on the fly, or a schedule cache respectively.

jerasure_invert_bitmatrix() inverts a $rows \times rows$ bit matrix, and the two following routines are helper routines for performing bit matrix dot products and scheduled operations respectively. Finally, **liberation_coding_bitmatrix** generates the Liberation Coding bit matrix defined in Section 3.3 for the given values of k and w .

6 Minimal Number of Ones

We state the following properties of RAID-6 codes and $w \times w$ bit matrices [4, 23]:

```

int ** jerasure_bitmatrix_to_schedule(int k, int m, int w, int *bitmatrix);
int *** jerasure_generate_schedule_cache(int k, int m, int w, int *bitmatrix);

void    jerasure_bitmatrix_encode(int k, int m, int w, int *bitmatrix,
                                char **data, char **coding,
                                int totalsize, int B);
void    jerasure_schedule_encode(int k, int m, int w, int **schedule,
                                char **data, char **coding,
                                int totalsize, int B);

int      jerasure_bitmatrix_decode(int k, int m, int w,
                                int *bitmatrix, int *erasures,
                                char **data, char **coding,
                                int totalsize, int B);
int      jerasure_schedule_decode_lazy(int k, int m, int w,
                                int *bitmatrix, int *erasures,
                                char **data, char **coding,
                                int totalsize, int B);
int      jerasure_schedule_decode_cache(int k, int m, int w,
                                int ***cache, int *erasures,
                                char **data, char **coding,
                                int totalsize, int B);

int      jerasure_invert_bitmatrix(int *mat, int *inv, int rows);
void     jerasure_bitmatrix_dotprod(int k, int w, int *bitmatrix_row,
                                int *src_ids, int dest_id,
                                char **data, char **coding,
                                int totalsize, int B);
void     jerasure_do_scheduled_operations(char **ptrs, int **schedule, int B);

int *    liberation_coding_bitmatrix(int k, int w);

```

Table 1: Relevant procedures from the Jerasure Coding Library [22].

- **Property #1:** Given a RAID-6 code that uses only XORs, this code may be represented by a CDM, which in turn may be specified by the matrices X_0, \dots, X_{k-1} . If the code is MDS, then each X_i must be an invertible $w \times w$ matrix.
- **Property #2:** Given a MDS RAID-6 code as above, for every i, j such that $i \neq j$, the matrix $(X_i + X_j)$ must be invertible.
- **Property #3:** If a $w \times w$ matrix is invertible, then it must have at least w ones.
- **Property #4:** A permutation matrix, I_π^w is a $w \times w$ matrix that has w ones such that there is exactly one one in every row and column of the matrix. Permutation matrices are the only matrices with exactly w ones that are invertible.
- **Property #5:** Let I_π^w and $I_{\pi'}^w$ be two permutation matrices. Their sum $(I_\pi^w + I_{\pi'}^w)$ is not invertible.

Now consider a RAID-6 code represented by X_0, \dots, X_{k-1} such that for some $i \neq j$, X_i and X_j have exactly w ones each. This code cannot be MDS, because X_i and X_j must be permutation matrices,

or they are not invertible. Since they are permutation matrices, their sum is not invertible. Therefore, a RAID-6 code may only have one X_i that has exactly w ones. The other X_j must have more than w ones.

Since the Liberation Codes have one matrix with exactly w ones, and $k-1$ matrices with $w+1$ ones, they are minimal RAID-6 matrices. It is an interesting by product of this argument that no MDS RAID-6 code can have optimal modification overhead. The Liberation Codes thereby achieve the lower bound on modification overhead. As an aside, the X-Code [29] does have optimal modification overhead; however the X-Code does not fit the RAID-6 paradigm.

7 Sketch of the MDS Proof

First we specify some notation: In the descriptions that follow, $\langle x \rangle_w$ is equal to $(x \bmod w)$. If x is negative, $\langle x \rangle_w$ is equal to $\langle w + x \rangle_w$.

It is a trivial matter to prove that the X_i matrices for Liberation codes are invertible. Moreover, it is easy to

prove that $(X_0 + X_i)$ is invertible for $0 < i < w$. The challenge in proving the Liberation codes to be MDS lies in the invertibility of $(X_i + X_j)$ for $0 < i < j < w$.

To demonstrate the invertibility of these matrices, we define a class of $w \times w$ matrices \mathcal{L} to contain all matrices M such that:

$$M = I_{\rightarrow i}^w + I_{\rightarrow j}^w + O_{x,x+i-1}^w + O_{z,z+j-1}^w,$$

with i, j, x and z being subject to the following constraints:

- w is an odd number greater than one.
- $0 \leq i < j < w$.
- $GCD(j-i, w) = 1$.
- If $(j-i)$ is even, $\langle z-x \rangle_w = w - \frac{j-i}{2}$.
- If $(j-i)$ is odd, $\langle z-x \rangle_w = \frac{w-(j-i)}{2}$.

In Figure 18 we show two examples matrices $\in \mathcal{L}$. In the first, $(j-i) = 4$, and $\langle z-x \rangle_w = 5$, which is indeed $7 - \frac{4}{2}$. In the second, $(j-i) = 1$, and $\langle z-x \rangle_w = 3$, which is $\frac{7-1}{2}$.

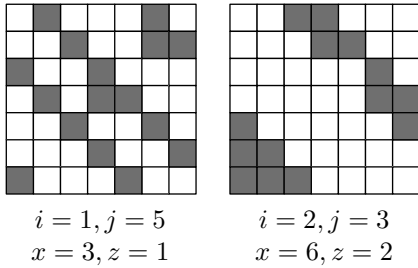


Figure 18: Two matrices $M \in \mathcal{L}$.

In the technical report, we prove by induction that all matrices $M \in \mathcal{L}$ are invertible [23]. Here we demonstrate that in the Liberation codes, when $0 < i < j < w$, the matrix $M = (X_i + X_j) \in \mathcal{L}$. For example, when $w = 7$, $(X_1 + X_5)$ is equal to the first matrix in Figure 18, and $(X_2 + X_3)$ is equal to the second matrix.

From the Liberation code definition in Section 3.3:

$$(X_i + X_j) = I_{\rightarrow i}^w + I_{\rightarrow j}^w + O_{x,x+i-1}^w + O_{z,z+j-1}^w,$$

where $x = \frac{i(w-1)}{2}$ and $z = \frac{j(w-1)}{2}$. Therefore, it is in the proper format to be an element of \mathcal{L} , so long as the constraints are satisfied.

Since w is prime number greater than two, it must be an odd number greater than one, and $GCD(j-i, w) = 1$. Now, consider the difference $(z-x)$:

$$\begin{aligned} (z-x) &= \frac{j(w-1)}{2} - \frac{i(w-1)}{2} \\ &= \frac{(j-i)(w-1)}{2}. \end{aligned}$$

When $(j-i)$ is even:

$$\begin{aligned} \langle z-x \rangle_w &= \left\langle \frac{(j-i)(w-1)}{2} \right\rangle_w \\ &= \left\langle \frac{j-i}{2}(w-1) \right\rangle_w \\ &= \left\langle -\frac{j-i}{2} \right\rangle_w \\ &= w - \frac{j-i}{2}. \end{aligned}$$

When $(j-i)$ is odd:

$$\begin{aligned} \langle z-x \rangle_w &= \left\langle \frac{(j-i)(w-1)}{2} \right\rangle_w \\ &= \left\langle \frac{(j-i-1)(w-1)}{2} + \frac{w-1}{2} \right\rangle_w \\ &= \left\langle w - \frac{j-i-1}{2} + \frac{w-1}{2} \right\rangle_w \\ &= \left\langle w + \frac{w-1-(j-i-1)}{2} \right\rangle_w \\ &= \left\langle w + \frac{w-(j-i)}{2} \right\rangle_w \\ &= \frac{w-(j-i)}{2}. \end{aligned}$$

Thus, $(X_i + X_j)$ fits the constraints to be an element of \mathcal{L} , and is invertible.

8 Conclusions/Future Work

In this paper, we have defined a new class of erasure codes, called Liberation Codes, for RAID-6 applications with k data devices. They are parity array codes represented by $w \times w$ bit matrices where w is a prime number $\geq k$. Their encoding performance is excellent, achieving a factor of $1 + \frac{2}{w}$ over optimal. This is an improvement in all cases over EVENODD encoding, and in some cases over RDP encoding. Their decoding performance does not outperform the other two codes, but has been measured to be within 15% of optimal. Their modification overhead is roughly two coding words per modified data word, which is not only an improvement over both EVENODD and RDP coding, but is fact optimal for a RAID-6 code.

In order to make decoding work quickly, we have presented an algorithm for scheduling the XOR operations of a bit matrix-vector product. The algorithm is simple and not effective for all bit matrices, but is very effective for Liberation decoding, reducing the overhead of decoding by a factor of six when $w = 17$, and over eleven when $w = 31$.

Besides comparing Liberation Codes to RDP and EVENODD coding, we assess their performance in com-

parison to Reed-Solomon coding. In all cases, they outperform Reed-Solomon coding greatly. We have written a freely-available library to facilitate the use of Liberation Codes in RAID-6 applications.

In sum, Liberation Codes are extremely attractive alternatives to other RAID-6 implementations. We anticipate that their simple structure, excellent performance and availability in library form will make them popular with RAID-6 implementors.

Our future work in this project is proceeding along three lines. First, the Liberation Codes are only defined for prime w . We are currently working to discover optimal RAID-6 codes for non-prime w . In particular, values of w which are powers of two are quite attractive. Our search has been based on Monte-Carlo techniques, attempting to build good matrices from smaller matrices and to improve on the best current matrices by modifying them slightly. Currently, the search has yielded optimal matrices for nearly every value of $k \leq 8$ and $w \leq 32$. We will continue to explore these constructions.

Second, we are looking to construct better bit matrix scheduling algorithms. Although the Liberation decoding cannot be improved much further, it is clear from our current algorithm's inability to schedule EVENODD coding effectively that further refinements are available. In its simplest case, bit scheduling is equivalent to common subexpression removal in compiler systems [1, 7, 8]. Huang *et al* have recently reduced this case to an NP-complete problem and give a heuristic based on matching to solve it [16]. However, the fact that one plus one equals zero in $GF(2)$ means that there are additional ways to improve performance, one of which is illustrated by the scheduling algorithm in Section 3.4. We are exploring these and other methodologies to further probe into the problem.

Finally, we have yet to explore how Liberation Codes may extrapolate systems that need to tolerate more failures. We plan to probe into minimal conditions for general MDS codes based on bit matrices such as those presented in Section 6, to see if the Liberation Code construction has application for larger classes of failures.

9 Availability

The **jerasure** library is available at <http://www.cs.utk.edu/~plank/plank/papers/CS-07-603.html>.

10 Acknowledgements

This material is based upon work supported by the National Science Foundation under grant CNS-0615221. The author would like to thank Adam Buchsbaum for

working to prove the Liberation Codes' MDS property, Lihao Xu for helpful discussions, and the program chairs and reviewers for their constructive comments.

References

- [1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [2] ANVIN, H. P. The mathematics of RAID-6. <http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf>, 2007.
- [3] BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. EVEN-ODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing* 44, 2 (February 1995), 192–202.
- [4] BLAUM, M., AND ROTH, R. M. On lowest density mds codes. *IEEE Transactions on Information Theory* 45, 1 (January 1999), 46–59.
- [5] BLAUM, M. M., BRADY, J. T., BRUCK, J., AND MENON, J. M. Method and means for encoding and rebuilding the data contents of up to two unavailable DASDs in a DASD array using simple non-recursive diagonal and row parity. U.S. Patent #5579475, 1996.
- [6] BLOMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An XOR-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.
- [7] BRIGGS, P., COOPER, K. D., AND SIMPSON, L. T. Value numbering. *Software: Practice & Experience* 27, 6 (1997), 701–724.
- [8] COCKE, J. Global common subexpression elimination. *ACM SIGPLAN Notices* 5, 7 (July 1970), 20–24.
- [9] CORBETT, P., ENGLISH, B., GOEL, A., GRACANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row diagonal parity for double disk failure correction. In *4th Usenix Conference on File and Storage Technologies* (San Francisco, CA, March 2004).
- [10] CORBETT, P. F., KLEIMAN, S. R., AND ENGLISH, R. M. Row-diagonal parity technique for enabling efficient recovery from double failures in a storage array. U.S. Patent #7203892, 2007.
- [11] FENG, G., DENG, R., BAO, F., AND SHEN, J. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers* 54, 9 (September 2005), 1071–1080.
- [12] FENG, G., DENG, R., BAO, F., AND SHEN, J. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥ 4) disk failures. *IEEE Transactions on Computers* 54, 12 (December 2005), 1473–1483.
- [13] HAFNER, J. L. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 211–224.
- [14] HAFNER, J. L. HoVer erasure codes for disk arrays. In *DSN-2006: The International Conference on Dependable Systems and Networks* (Philadelphia, June 2006), IEEE.
- [15] HUANG, C., CHEN, M., AND LI, J. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. In *NCA-07: 6th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2007).
- [16] HUANG, C., LI, J., AND CHEN, M. On optimizing XOR-based codes for fault-tolerant storage applications. In *ITW'07, Information Theory Workshop* (Tahoe City, CA, September 2007), IEEE, pp. 218–223.

- [17] HUANG, C., AND XU, L. STAR: An efficient coding scheme for correcting triple storage node failures. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 197–210.
- [18] MACWILLIAMS, F. J., AND SLOANE, N. J. A. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.
- [19] MOON, T. K. *Error Correction Coding: Mathematical Methods and Algorithms*. John Wiley & Sons, New York, 2005.
- [20] PETERSON, W. W., AND WELDON, JR., E. J. *Error-Correcting Codes, Second Edition*. The MIT Press, Cambridge, Massachusetts, 1972.
- [21] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience* 27, 9 (September 1997), 995–1012.
- [22] PLANK, J. S. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Tech. Rep. CS-07-603, University of Tennessee, September 2007.
- [23] PLANK, J. S., AND BUCHSBAUM, A. L. Some classes of invertible matrices in $GF(2)$. Tech. Rep. CS-07-599, University of Tennessee, August 2007.
- [24] PLANK, J. S., AND DING, Y. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience* 35, 2 (February 2005), 189–194.
- [25] PLANK, J. S., AND XU, L. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *NCA-06: 5th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2006).
- [26] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics* 8 (1960), 300–304.
- [27] VAN LINT, J. H. *Introduction to Coding Theory*. Springer-Verlag, New York, 1982.
- [28] WYLIE, J. J., AND SWAMINATHAN, R. Determining fault tolerance of XOR-based erasure codes efficiently. In *DSN-2007: The International Conference on Dependable Systems and Networks* (Edinburgh, Scotland, June 2007), IEEE.
- [29] XU, L., AND BRUCK, J. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory* 45, 1 (January 1999), 272–276.

Are Disks the Dominant Contributor for Storage Failures?

A Comprehensive Study of Storage Subsystem Failure Characteristics

Weiham Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky[†]

Department of Computer Science,
University of Illinois at Urbana Champaign
{wjiang3,chu7,yyzhou}@cs.uiuc.edu

[†]Network Appliance, Inc.
arkady@netapp.com

Abstract

Building reliable storage systems becomes increasingly challenging as the complexity of modern storage systems continues to grow. Understanding storage failure characteristics is crucially important for designing and building a reliable storage system. While several recent studies have been conducted on understanding storage failures, almost all of them focus on the failure characteristics of one component – disks – and do not study other storage component failures.

This paper analyzes the failure characteristics of storage subsystems. More specifically, we analyzed the storage logs collected from about 39,000 storage systems commercially deployed at various customer sites. The data set covers a period of 44 months and includes about 1,800,000 disks hosted in about 155,000 storage shelf enclosures. Our study reveals many interesting findings, providing useful guideline for designing reliable storage systems. Some of our major findings include: (1) In addition to disk failures that contribute to 20-55% of storage subsystem failures, other components such as physical interconnects and protocol stacks also account for significant percentages of storage subsystem failures. (2) Each individual storage subsystem failure type and storage subsystem failure as a whole exhibit strong self-correlations. In addition, these failures exhibit “bursty” patterns. (3) Storage subsystems configured with redundant interconnects experience 30-40% lower failure rates than those with a single interconnect. (4) Spanning disks of a RAID group across multiple shelves provides a more resilient solution for storage subsystems than within a single shelf.

1 Introduction

1.1 Motivation

Reliability is a critically important issue for storage systems because storage failures can not only cause ser-

vice downtime, but also lead to data loss. Building reliable storage systems becomes increasingly challenging as the complexity of modern storage systems grows into an unprecedented level. For example, the EMCTM Symmetrix DMX-4 can be configured with up to 2400 disks [8], the GoogleTM File System cluster is composed of 1000 storage nodes [9], and the NetApp[®] FAS6000 series can support more than 1000 disks per node, with up to 24 nodes in a system [12].

To make things even worse, disks are not the only component in storage systems. To connect and access disks, modern storage systems also contain many other components, including shelf enclosures, cables and host adapters, and complex software protocol stacks. Failures in these components can lead to downtime and/or data loss of the storage system. Hence, in complex storage systems, component failures are very common and critical to storage system reliability.

To design and build a reliable storage system, it is crucially important to understand the storage failure characteristics. First, accurate estimation of storage failure rate can help system designers decide how many resources should be used to tolerate failures and to meet certain service-level agreement (SLA) metrics (*e.g.*, data availability). Second, knowledge about factors that greatly impact the storage system reliability can guide designers to select more reliable components or build redundancy into unreliable components. Third, understanding the statistical properties such as failure distribution over time of modern storage systems is necessary to build right testbed and fault injection models to evaluate existing resiliency mechanisms and to develop better fault-tolerant mechanisms.

While several recent studies have been conducted on understanding storage failures, almost all of them focused on the failure characteristics of one storage component—disks. For example, disk vendors have studied the disk failure characteristics through running accelerated life tests and collecting statistics from their return unit databases [4, 21]. Based on such tests, they calculate

the *mean-time-to-failure (MTTF)* and record it in a disk specification. For most of the disks, the specified MTTF is typically more than one million hours, equivalent to a lower than 1% *annualized failure rate (AFR)*. But such low AFR is usually not what has been experienced by users. Motivated from this observation, recently some researchers have studied *disk failures* from a user's perspective by analyzing disk replacement logs collected in the field [14, 16]. Interestingly, they found disks are replaced much more frequently (2-4 times) than vendor-specified AFRs. But as this study indicates, there are other storage subsystem failures besides disk failures that are treated as disk faults and lead to unnecessary disk replacements. Additionally, some researchers analyzed the characteristics of disk sector errors, which can potentially lead to complete disk failures [2], and they found that sector errors exhibit strong temporal locality (*i.e.*, bursty patterns).

While previous works provide very good understanding of disk failures and an inspiring starting point, it is not enough since, besides disks, there are many other components that may contribute to storage failures. Without a good understanding of these components' failure rates, failure distributions, and other characteristics, as well as impacts of these component failures on the storage system, it can make our estimation of the storage failure rate/distribution inaccurate. For example, as we will show in our study from real-world field data, having a lower disk failure rate does not necessarily mean that the corresponding storage system is more reliable—because some other components may not be as reliable.

More importantly, if we only focus on disk failures and ignore other component failures, we may fail to build a highly reliable storage system. For example, RAID is usually the only resiliency mechanism built in to most modern storage systems (various forms of checksumming are considered as part of RAID). As RAID is mainly designed to tolerate disk failures, it is insufficient to handle other component failures such as failures in shelf enclosures, interconnects, and software protocol layers.

While we are interested in failures of a whole storage system, this study is concentrated on the core part of it — the *storage subsystem*, which contains disks and all components providing connectivity and usage of disks to the entire storage system.

We conducted a study using real-world field data from Network ApplianceTM AutoSupport Database, to answer the following questions:

- How much do disk failures contribute to storage subsystem failures? What are other major factors that can lead to storage subsystem failures?
- What are the failure rates of other types of storage subsystem components such as physical interconnects and protocol stacks? What are the failure

characteristics such as failure distribution and failure correlation for these components?

- Typically, some resiliency mechanisms such as RAID and redundancy mechanisms such as multi-pathing are used in practice to achieve high reliability and availability [5, 9]. How effective are these mechanisms in handling storage subsystem failures?

Data from the same AutoSupport Database was first analyzed in [2] on latent sector errors and was further analyzed in [3] on data corruptions.

There are other redundancy and resiliency mechanisms in storage system layers higher than the storage subsystem and RAID-based resiliency mechanism studied in this paper. These mechanisms handle some of the storage subsystem failures. Studying impacts of these resiliency and redundancy mechanisms on storage failures, including storage subsystem failures, is part of the future work.

1.2 Our Contributions

This paper analyzes the failure characteristics of storage subsystems, including disks and other system components, based on a significant amount of field data collected from customers. Specifically, we analyzed the storage logs collected from about 39,000 storage systems commercially deployed at various customer sites. The data set covers a period of 44 months and includes about 1,800,000 disks hosted in about 155,000 storage shelf enclosures. Furthermore, our data covers a wide range of storage system classes, including *near-line (backup)*, *low-end*, *mid-range*, and *high-end* systems.

This paper studies failure characteristics from several angles. First, we classify storage subsystem failures into four failure types based on their symptoms and root causes and examine the relative frequency of each failure type. Second, we study the effect of several factors on storage subsystem reliability. These factors include disk models, shelf enclosure models, and network redundancy mechanisms. Finally, we analyze the statistical properties of storage subsystem failures, including the correlation between failures and their time distribution.

Our study reveals many interesting findings, providing useful guideline for designing reliable storage systems. Following is a summary of our major findings and the corresponding implications:

- In addition to disk failures that contribute to 20-55% of storage subsystem failures, other components such as physical interconnects (including shelf enclosures) and protocol stacks also account for significant percentages (27-68% and 5-10%, respectively) of failures. Due to these component failures, even though storage systems of certain types (*e.g.*,

low-end primary systems) use more reliable disks than some other types (e.g., near-line backup systems), their storage subsystems exhibit higher failure rates. These results indicate that, to build highly reliable and available storage systems, only using resiliency mechanisms targeting disk failures (e.g., RAID) is not enough. We also need to build resiliency mechanisms such as redundant physical interconnects and self-checking protocol stacks to tolerate failures in these storage components.

- Each individual storage subsystem failure type and storage subsystem failure as a whole exhibit strong correlations, (*i.e.* after one failure, the probability of additional failures of the same type is higher). In addition, failures also exhibit bursty patterns in time distribution, (*i.e.* multiple failures of the same type tend to happen relatively close together). These results motivate a revisiting of current resiliency mechanisms such as RAID that assume independent failures. These results also motivate development of better resiliency mechanisms that can tolerate multiple correlated failures and bursty failure behaviors.
- Storage subsystems configured with two independent interconnects experienced much (30-40%) lower AFRs than those with a single interconnect. This result indicates the importance of interconnect redundancy in the design of reliable storage systems.
- RAID groups built with disks spanning multiple shelf enclosures show much less bursty failure patterns than those built with disks from the same shelf enclosure. This indicates that the former is a more resilient solution for large storage systems.

The rest of the paper is organized as follows. Section 2 provides the background and describes our methodology. Section 3 presents the contribution of disk failures to storage subsystem failures and frequency of other types of storage subsystem failures. Section 4 quantitatively analyzes the effects of several factors on storage subsystem reliability, while Section 5 analyzes the statistical properties of storage subsystem failures. Section 6 discusses the related work, and Section 7 concludes the paper and provides directions for future work.

2 Background and Methodology

In this section, we detail the typical architecture of storage systems, the definition and terminology used in this paper, and the source of the data studied in this paper.

2.1 Storage System Architecture

Figure 1 shows the typical architecture of a modern storage system.

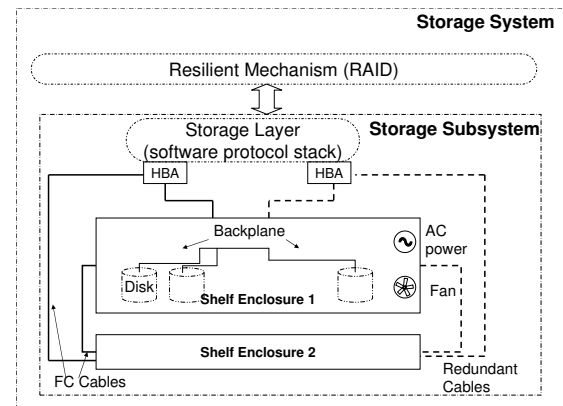


Figure 1. Storage system architecture.

From the customers' perspective, a storage system is a virtual device that is attached to customers' systems and provides customers with the desired storage capacity with high reliability, good performance, and flexible management.

Looking from inside, a storage system is composed of storage subsystems, resiliency mechanisms, storage head/controller, and other higher-level system layers. The storage subsystem is the core part of a storage system and provides connectivity and usage of disks to the entire storage system. It contains various components, including disks, shelf enclosures, cables and host adapters, and complex software protocol stacks. Shelf enclosures provide power supply, cooling service and prewired backplane for the disks mounted in them. Cables initiated from host adapters connect one or multiple shelf enclosures to the network. Each shelf enclosure can be optionally connected to a secondary network for redundancy. In Section 4.3 we will show the impact of this redundancy mechanism on failures of the storage subsystem.

Usually, on top of the storage subsystem, resiliency mechanisms, such as RAID, are used to tolerate failures in storage subsystems.

2.2 Terminology

We use the followings terms in this paper.

- **Disk family:** A particular disk product. The same product may be offered in different capacities. For example, "Seagate Cheetah 10k.7" is a disk family.
- **Disk model:** The combination of a disk family and a particular disk capacity. For example, "Seagate Cheetah 10k.7 300 GB" is a disk model. For disk family and disk model, we use the same naming convention as in [2, 3].
- **Failure types:** Refers to the four types of storage subsystem failures: disk failure, physical intercon-

nect failure, protocol failure, and performance failure.

- **Shelf enclosure model:** A particular shelf enclosure product. All shelf enclosure models studied in this paper can host at most 14 disks.
- **Storage subsystem failure:** Refers to failures that prevent the storage subsystem from providing storage service to the whole storage system. However, not all storage subsystem failures are experienced by customers, since some of the failures can be handled by resiliency mechanisms on top of storage subsystems (e.g. RAID) and other mechanisms at higher layers.
- **Storage system class:** Refers to the capability and usage of storage systems. There are four storage system classes studied in this paper: near-line systems (mainly used as secondary storage), low-end, mid-range, and high-end (mainly used as primary storage).
- Other terms in the paper are used as defined by SNIA [19].

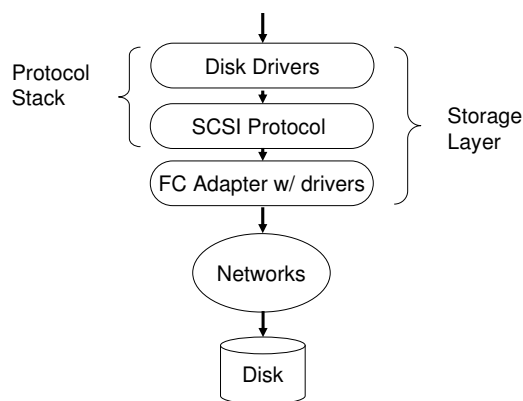


Figure 2. I/O request path in storage subsystem

2.3 Definition and Classification of Storage Subsystem Failures

Figure 2 shows the steps and components that are involved in fulfilling an I/O request in a storage subsystem. As shown in Figure 2, for the storage layer to fulfill an I/O request, the I/O request will first be processed and transformed by protocols and then delivered to disks through

networks initiated by host adapters. *Storage subsystem failures* are the failures that break the I/O request path, and can be caused by hardware failures, software bugs, and protocol incompatibilities along the path.

To better understand storage subsystem failures, we partition them into four categories along the I/O request path:

- **Disk Failure:** This type of failure is triggered by failure mechanisms of disks. Imperfect media, media scratches caused by loose particles, rotational vibration, and many other factors internal to a disk can lead to this type of failures. Sometimes, the storage layer proactively fails disks based on statistics collected by on-disk health monitoring mechanisms (e.g., a disk has experienced too many sector errors [1]. These incidences are also counted as *disk failures*).
- **Physical Interconnect Failure:** This type of failure is triggered by errors in the networks connecting disks and storage heads. It can be caused by host adapter failures, broken cables, shelf enclosure power outage, shelf backplanes errors, and/or errors in shelf FC drivers. When *physical interconnect failures* happen, affected disks appear to be missing from the system.
- **Protocol Failure:** This type of failure is caused by incompatibility between protocols in disk drivers or shelf enclosures and storage heads and software bugs in the disk drivers. When this type of failure happens, disks are visible to the storage layer but I/O requests are not correctly responded by disks.
- **Performance Failure:** This type of failure happens when the storage layer detects that a disk cannot serve I/O requests in a timely manner while none of previous three types of failures are detected. It is mainly caused by partial failures, such as unstable connectivity or when disks are heavily loaded with disk-level recovery (e.g., broken sector remapping).

The occurrences of these four types of failures are recorded in logs collected by Network Appliance.

2.4 Data Sources

Table 1 provides an overview of the data used in this study. Support logs from about 39,000 commercially deployed storage systems in four system classes are used for the results presented in this paper. There are totally about 1,800,000 disks mounted in 155,000 shelf enclosures. The disks are a combination of SATA and FC disks. The population of disks contains at least 9 disk families and 15 disk models. The storage logs used for

System Classes	Duration	# Systems	# Shelves	Multipathing	# Disks	Disk Types	# RAID Groups	RAID Types	# Failure Types	# Failure Events
Near-line (Backup)	1/04 - 8/07	4,927	33,681	single path	520,776	SATA	67,227	RAID4 RAID6	Disk Failure Physical Inter. Failure Protocol Failure Performance Failure	10,105 4,888 1,819 1,080
Low-end	1/04 - 8/07	22,031	37,260	single-path	264,983	FC	44,252	RAID4 RAID6	Disk Failure Physical Inter. Failure Protocol Failure Performance Failure	3,230 4,338 1,021 1,235
Mid-range	1/04 - 8/07	7,154	52,621	single-path dual-path	578,980	FC	77,831	RAID4 RAID6	Disk Failure Physical Inter. Failure Protocol Failure Performance Failure	8,989 7,949 2,298 2,060
High-end	1/04 - 8/07	5,003	33,428	single-path dual-path	454,684	FC	49,555	RAID4 RAID6	Disk Failure Physical Inter. Failure Protocol Failure Performance Failure	8,240 7,395 1,576 153

Table 1. Overview of studied storage systems. Note that the “# Disks” given in the table is the number of disks that have ever been installed in the system during the 44 months. For some systems, disks have been replaced during the period, and we account for that in our analysis by calculating the life time of each individual disk. The “# Failure Events” given in the table are the numbers of the four types of storage subsystem failures (disk failure, physical interconnect failure, protocol failure, and performance failure) that happened during the period.

this study were collected between January 2004 and August 2007.

Below we describe each storage system class.

Near-line systems are deployed as cost-efficient archival or backup storage systems. Less expensive SATA disks are used in nearline systems. In nearline systems, one storage subsystem on average contains about 7 shelf enclosures and 98 disks. Both RAID4 and RAID6 are supported as resiliency mechanisms in nearline systems.

Primary storage systems, including low, mid, and high-end systems, are mainly used in mission- or business-critical environments and primarily use FC disks. Low-end storage systems have embedded storage heads with shelf enclosures, but external shelf enclosures can be added. Mid-range and high-end systems use external shelves and are usually configured with more shelf enclosures and disks than low-end systems. Each mid-range system has about 7 shelf enclosures and 80 disks (not every shelf is fully utilized and configured with 14 disks), and high-end systems are in similar scale. Going from low to high-end systems, more reliable components and more redundancy mechanisms are used. For example, both mid-range and high-end systems support dual paths for redundant connectivity.

2.5 Support Logs and Analysis

The storage systems studied in this paper have a low-overhead logging mechanism that automatically records informational and error events on each layer (software and hardware) and each subsystem during operation. Several recent works such as [2, 3] also studied the same set of storage logs from different aspects.

Figure 3 shows a log example that reports a physical

interconnect failure. As can be seen in the figure, when a failure happens, multiple events are generated as the failure propagates from lower layers to higher layers (Fibre Channel to SCSI to RAID). By keeping track of events generated by lower layers, higher layers can identify the cause of events and tag the events with corresponding failure types. In this example, the RAID layer, which is right above the storage subsystem, generates a disk missing event, indicating a physical interconnect failure. In this paper, we look at four types of events generated by the RAID layer, corresponding to four categories of storage subsystem failures.

Besides the events shown in the example, there are many other events recorded in the logs. For example, standard error reports from the SCSI protocol layer tell us what failure mechanisms happen inside disks [18]. Disk medium error messages from disk drivers provide information about broken sectors [2]. Similarly, messages from FC protocol and FC host adapter drivers report errors that occur in FC networks and FC adapters.

It is important to notice that not all failures propagate to the RAID layer, as some failures are recovered or tolerated by storage subsystems. For example, an interconnect failure can be recovered through retries at SCSI layer or be tolerated through multipathing. Therefore, storage failures characterized as storage subsystem failure as a whole are those errors exposed by storage subsystems to the rest of the system.

As Figure 3 shows, each event is tagged with the timestamps when the failure is detected and with the ID of the disk affected by the failure. Since all the storage systems studied in this paper periodically send data verification requests to all disks as a proactive method to detect failures, the lag between the occurrence and the detection of the failure is usually shorter than an hour.

System information is also copied with snapshots and recorded in storage logs on a weekly basis. This information is particularly important for understanding storage subsystem reliability since it provides the insight into the system parameters of storage subsystems. More specifically, storage logs contain the information about hardware components used in storage subsystems, such as disk models and shelf enclosure models, and they also contain the information about the layout of disks, such as which disks are mounted in the same shelf enclosures, and which disks are in the same RAID group. This information is used for analyzing statistical properties of storage subsystem failures in Section 5.

3 Frequency of Storage Subsystem Failures

As we categorize storage subsystem failures into four failure types based on their root causes, a natural question is therefore what the relative frequency of each failure type is. To answer this question, we study the storage logs collected from 39,000 storage systems.

Figure 4(a) presents the breakdown of AFR for storage subsystems based on failure types, for all four system classes studied in this paper. Since one problematic disk family, denoted as *Disk H*, has already been reported in [2], for Figure 4(b) we exclude data from storage subsystems using *Disk H*, so that we can analyze the trend without being skewed by one problematic disk family. The discussion on *Disk H* is presented in Section 4.1.

Finding (1): In addition to *disk failures* (20-55%), *physical interconnect failures* make up a significant part (27-68%) of storage subsystem failures. *Protocol failures* and *performance failures* both make up noticeable fractions.

Implications: *Disk failures* are not always a dominant factor of storage subsystem failures, and a reliability study for storage subsystems cannot only focus on *disk failures*. Resilient mechanisms should target all failure types.

- Sun Jul 23 05:43:36 PDT [fc.device.timeout:error]: Adapter 8 encountered a device timeout on device 8.24
- Sun Jul 23 05:43:50 PDT [fc.adapter.reset:info]: Resetting Fibre Channel adapter 8.
- Sun Jul 23 05:43:50 PDT [scsi.cmd.abortedByHost:error]: Device 8.24: Command aborted by host adapter.
- Sun Jul 23 05:44:12 PDT [scsi.cmd.selectionTimeout:error]: Device 8.24: Adapter/target error: Targeted device did not respond to requested I/O. I/O will be retried.
- Sun Jul 23 05:44:22 PDT [scsi.cmd.noMorePaths:error]: Device 8.24: No more paths to device. All retries have failed.
- Sun Jul 23 05:46:22 PDT [raid.config.filesystem.disk.missing:info]: File system Disk 8.24 S/N [3EL03PAV0000711LR8W] is missing.

Figure 3. Example of a piece of log reporting a physical interconnect failure.

As Figure 4(b) shows, across all system classes, *disk failures* do not always dominate storage subsystem failures. For example, in low-end storage systems, the AFR for storage subsystems is about 4.6%, while the AFR for *disks* is only 0.9%, about 20% of overall AFR. On the other hand, *physical interconnect failures* account for a significant fraction of storage subsystem failures, ranging from 27% to 68%. The other two failure types, *protocol failures* and *performance failures*, contribute to 5-10% and 4-8% of storage subsystem failures, respectively.

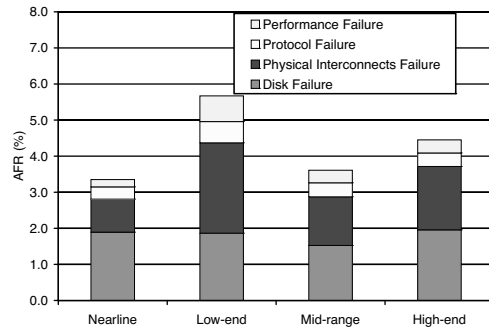
Finding (2): For *disks*, near-line storage systems show higher (1.9%) AFR than low-end storage systems (0.9%). But for the whole *storage subsystem*, near-line storage systems show lower (3.4%) AFR than low-end storage systems (4.6%).

Implications: *Disk failure rate* is not indicative of the storage subsystem failure rate.

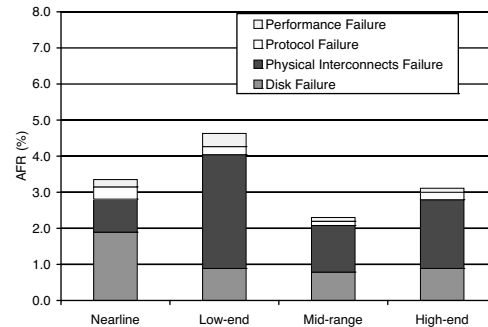
Figure 4(b) also shows that near-line systems, which mostly use SATA disks, experience about 1.9% AFR for *disks*, while for low-end, mid-range, and high-end systems, which mostly use FC disks, the AFR for *disks* is under 0.9%. This observation is consistent with the common belief that enterprise disks (FC) are more reliable than near-line disks (SATA).

However, the AFR for storage subsystems does not follow the same trend. Storage subsystem AFR of near-line systems is about 3.4%, lower than that of low-end systems (4.6%). This indicates that other factors, such as shelf enclosure model and network configurations, strongly affect storage subsystem reliability. The impacts of these factors are examined in the next section.

Another interesting observation that can be seen in Figure 4(b) is that for FC drives, the *disk failure rate* is consistently below 1%, as published by disk drive manufacturers, while some previous works claim that the AFR for disks is much higher [14, 16]. We believe that the main reason for the discrepancy is that these studies look at disk failures from different angles. Our study is from a system's perspective, as we extract disk failure events from system logs, similar to disk drive manufacturers' studies. On the other hand [14, 16], look at disk failures from a user's perspective. Since their studies are based on disk replacement logs, they cannot identify the reasons for disk replacement. As system administrators often replace disks when they observe unavailability of disks, the disk replacement rates reported in these studies are actually close to the *storage subsystem failure rate* of this paper.



(a) Including storage subsystems using *Disk H*



(b) Excluding storage subsystems using *Disk H*

Figure 4. AFR for storage subsystems in four system classes and the breakdown based on failure types.

4 Impact of System Parameters on Storage Subsystem Failures

As we have seen above, storage subsystems of different system classes show different AFRs. While these storage subsystems are architecturally similar, the characteristics of their components, like disks and shelves, and their redundancy mechanisms, like multipathing, differ. We now explore the impact of these factors on storage subsystem failures.

4.1 Disk Model

The disk is the key component of a storage subsystem; therefore it is important to understand how disk models affect storage subsystem failures. To understand the impact of the disk model, we study data collected from nearline, low-end, mid-range, and high-end systems.

Figure 5 shows the AFRs for storage subsystems from 4 system classes configured with 3 shelf enclosure models, 6 combinations in total (not every shelf enclosure model works with all system classes). Since we find that the enclosure model also has a strong impact on storage subsystem failures, we group data based on system class, shelf enclosure model, and disk model so that we can separately study the effects of these factors. In this section, we mainly focus on disk model; shelf enclosure model will be discussed in Section 4.2.

There are a total of 20 disk models used in these systems, and each disk model is denoted as *family-type*, with the same convention as in [2]. For anonymization purpose, a single letter is used to represent a disk family (e.g., Seagate Cheetah 10k.7), and type is a single number indicating the disk's capacity. The relative capacity within a family is ordered by the number. For example, *Disk A-2* is larger than *A-1* and *B-2* is larger than *B-1*.

Finding (3): Storage subsystems using disks from a problematic disk family show much higher (2 times) AFR than other storage subsystems.

Implications: Disk model is a critical factor to consider for designing reliable storage subsystems.

We can see from Figure 5 (a)-(f) that for most storage subsystems, AFR is about 2% - 4%. However, storage subsystems using *Disk H-1* and *Disk H-2* show 3.9%-8.3% AFR, higher than the average AFR by a factor of two.

We know that *Disk H-1* and *Disk H-2* are problematic. It is interesting to observe that not only disk failures but also protocol failures and performance failures are negatively affected by the problematic disks. The possible reason is that as disks experience failures, corner-case bugs in the protocol stacks are more likely to be triggered, leading to more occurrences of protocol failures. At the same time, some I/O requests cannot be served in time, causing more performance failures.

Finding (4): Storage subsystems using disks from the same disk models exhibit similar *disk failure* rates across different system environments (different system class or shelf enclosure models), but they show very different *storage subsystem failure* rates.

Implications: Factors other than disk models also heavily affect storage subsystem failures, while they are not revealed by *disk failures*.

As Figure 5 shows, some disk models are used by storage subsystems of multiple system classes, together with various shelf enclosure models. For example, *Disk A-2*

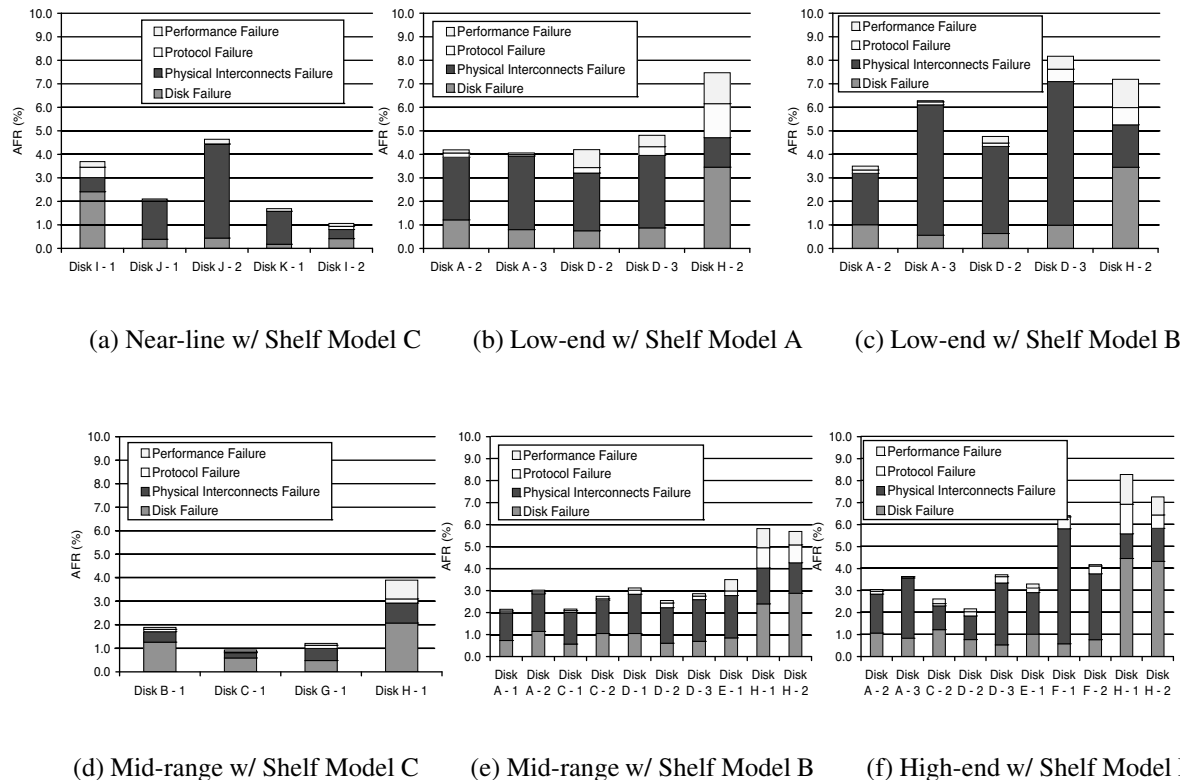


Figure 5. AFR for storage subsystems by disk models.

and *Disk D-2* are used in low-end systems with different shelf models and by mid-range and high-end systems with the same shelf model.

As we can see from Figure 5, for the storage subsystems using the same disk models, *disk failure* rates do not change much. For example, *disk AFR* of *Disk D-2* varies from 0.6% to 0.77% with a standard deviation of 8%. For all storage subsystems sharing the same disk models, the average standard deviation of *disk AFR* is less than 11%.

On the other hand, the *storage subsystem AFR* exhibits strong variation. For example, AFR for *storage subsystems* using *Disk D-2* varies from 2.2% to 4.9%, with a standard deviation of 127%. For all such storage subsystems, the average standard deviation of *storage subsystem AFR* is as high as 98%. This observation indicates that *storage subsystem AFR* is strongly affected by factors other than *disk model*, while these factors do not affect *disk failures* much.

using *Disk D-2* show lower *disk* and *storage subsystem AFR* than those using *Disk D-1*.

4.2 Shelf Enclosure Model

Shelf enclosures contain power supplies, cooling devices, and prewired backplanes that carry power and I/O bus signals to the disks mounted in them. Different shelf enclosure models are different in design and have different mechanisms for providing these services; therefore, it is interesting to see how shelf enclosure model affects storage subsystem failures.

In order to study the impact of the shelf enclosure model, we look at the data collected from low-end storage systems, since low-end systems use the same disk models with different shelf enclosure models, so that we can study the effect of shelf enclosure models without inference from disk models.

Finding (5): The AFR for *disks* and *storage subsystems* does not increase with disk size.

Implications: As disk capacity rapidly increases, storage subsystems will not necessarily experience more *disk failures* or *storage subsystem failures*.

We do not observe increasing *disk failure* rate or *storage subsystem failure* rate with increasing disk capacity. For example, as Figure 5 (e) shows, storage subsystems

Finding (6): The shelf enclosure model has a strong impact on storage subsystem failures, and different shelf enclosure models work better with different disk models.

Implications: To build a reliable storage subsystem, hardware components other than disks (*e.g.*, shelf enclosure) should also be carefully selected. And due to component interoperability issues, there might be a different “best choice” for one component depending on the choice of other components.

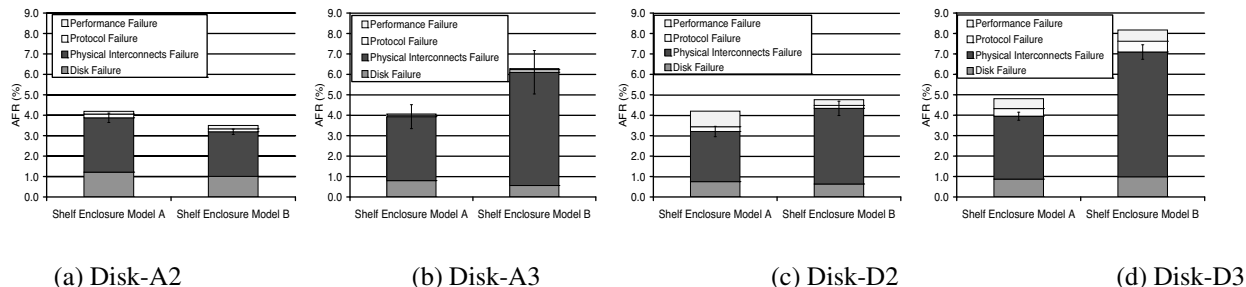


Figure 6. AFR for storage subsystems of low-end storage systems by shelf enclosure models using the same disk models (a subset of data from Figure 5). The error bars show 99.5%+ confidence intervals for physical interconnect failures.

Figure 6 (a)-(d) shows AFR for storage subsystems when configured with different shelf enclosure models but the same disk models. As expected, shelf enclosure model primarily impacts *physical interconnect failures*, with little impact on other failure types, different from disk model, which impacts all failure types.

To confirm this observation, we tested the statistical significance using a T-test [15]. As Figure 6 (a) shows, the *physical interconnect failures* with different shelf enclosure models are quite different ($2.66 \pm 0.23\%$ versus $2.18 \pm 0.13\%$). A T-test shows that this is significant at the 99.5% confidence interval, indicating that the hypothesis that *physical interconnect failures* are impacted by shelf enclosure models is very strongly supported by the data. Figure 6(b)-(d) shows similar observations with significance at 99.5%, 99.9%, and 99.9% confidence.

It is also interesting to observe that for different disk models, different shelf enclosure models work better. For example, for *Disk-A2*, storage subsystems using *Shelf Enclosure B* show better reliability than those using *Shelf Enclosure A*, while for *Disk-A3*, *Disk-D2*, and *Disk-D3*, *Shelf Enclosure A* is more reliable. Such observations might be due to component interoperability issues between disks and shelf enclosures. This indicates that we might not be able to make the best decision on selecting the most reliable hardware components without evaluating the components from a system perspective and taking the effect of interoperability into account.

4.3 Network Redundancy Mechanism

As we have seen, *physical interconnect failures* contribute to a significant fraction (27-68%) of storage subsystem failures. Since *physical interconnect failures* are mainly caused by network connectivity issues in storage subsystems, it is important to understand the impact of network redundancy mechanisms on storage subsystem failures.

For the mid-range and high-end systems studied in this

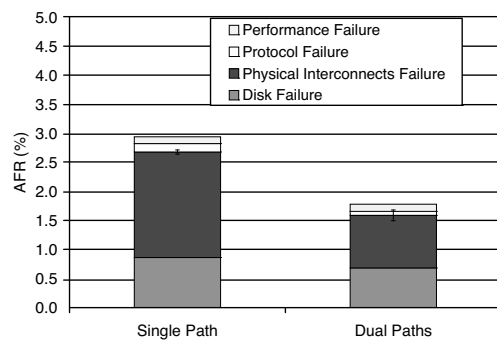
paper, FC drivers support a network redundancy mechanism, commonly called *active/passive multipathing*. This network redundancy mechanism connects shelves to two independent FC networks, and redirects I/O requests through the redundant FC network when one FC network experiences network component failures (e.g., broken cables).

To study the effect of this network redundancy mechanism, we look at the data collected from mid-range and high-end storage systems, and group them based on whether the network redundancy mechanism is turned on. As we observed from our data set, about 1/3 of storage subsystems are utilizing the network redundancy mechanism, while the other 2/3 are not. We call these two groups of storage subsystems *dual paths* systems and *single path* systems, respectively. In our data set, there are very few disk models used in both configurations; other disk models are mainly used in either *dual paths* systems or *single path* systems. Therefore, we cannot further break down the results based on disk models and shelf enclosure models.

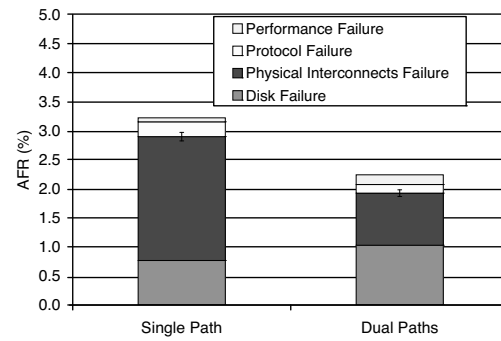
Finding (7): Storage subsystems configured with network redundancy mechanisms experience much lower (30-40% lower) AFR than other systems. AFR for *physical interconnects* is reduced by 50-60%.

Implications: Network redundancy mechanisms such as multipathing can greatly improve the reliability of storage subsystems.

Figure 7 (a) and (b) show the AFR for storage subsystems in mid-range and high-end systems, respectively. As expected, secondary path reduces *physical interconnect failures* by 50-60% ($1.82 \pm 0.04\%$ versus $0.91 \pm 0.09\%$ and $2.13 \pm 0.07\%$ versus $0.90 \pm 0.06\%$), with little impact on other failure types. Since *physical interconnect failure* is just a subset of all *storage subsystem failures*, AFR for *storage subsystems* is reduced by 30-40%. This



(a) Mid-range systems



(b) High-end systems

Figure 7. AFR for storage subsystems broken down by the number of paths. The error bars show 99.9% confidence intervals for physical interconnect failures.

indicates that multipathing is an exceptionally good redundancy mechanism that delivers reduction of failure rates as promised. As we applied a T-test on these results, we found out that for both mid-range and high-end systems the observation is significant at the 99.9% confidence interval, indicating that the data strongly support the hypothesis that physical interconnect failures are reduced by multipathing configuration.

However, the observation also tells us that there is still further potential in network redundancy mechanism designs. For example, given that the probability for one network to fail is about 2%, the idealized probability for two networks to both fail should be a few magnitudes lower (about 0.04%). But the AFR we observe is far from the ideal number.

One reason is that not only failures from networks between shelves contribute to *physical interconnect failures*; other failures, such as shelf backplane errors, can also lead to *physical interconnect failures*, while multipathing does not provide redundancy for shelf backplane. Another possible reason is that most modern host adapters support more than one port, and each port can be used as a “logical” host adapter. If two independent networks are initiated by two “logical” host adapters sharing the same physical host adapter, a host adapter failure can cause failures of both networks.

5 Statistical Properties of Storage Subsystem Failures

An important aspect of storage subsystem failures is their statistical properties. Understanding the statistical properties such as failure distribution of modern storage subsystems is necessary to build right testbed and fault injection models to evaluate existing resiliency mecha-

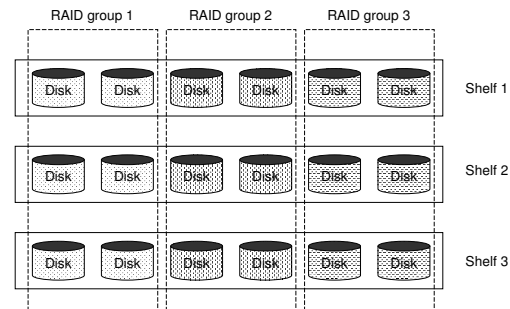


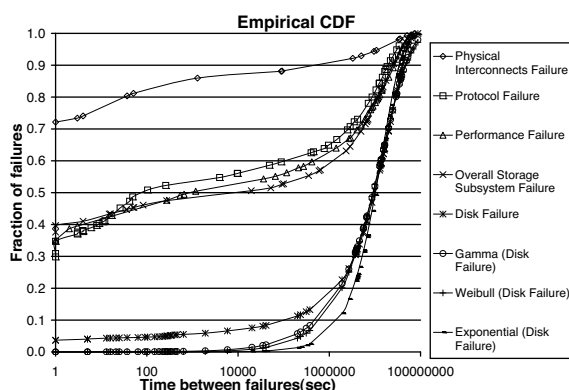
Figure 8. Disk layout in shelf enclosures and RAID groups.

nisms and to develop better ones. For example, some researchers have assumed a constant failure rate, which means an exponentially distributed time between failures, and that failures are independent, when calculating the expected time to failure for a RAID [13].

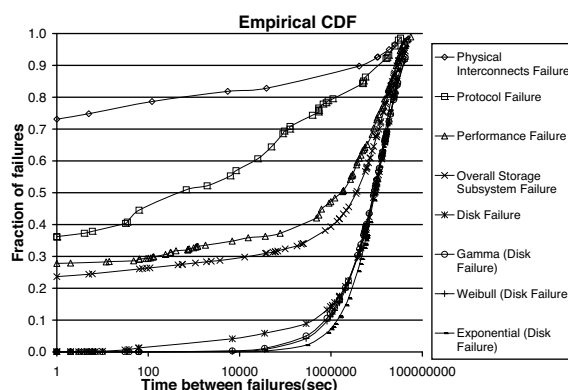
Figure 8 illustrates how disks are laid out in storage subsystems. As Figure 8 shows, multiple disks are mounted in one shelf enclosure and share the cooling service, power supply, and intrashelf connectivity provided by the shelf enclosure.

The figure also shows how disks are assigned to build up RAID groups, which include both data disks and parity disks containing redundant data. In order to prevent shelf enclosure from being the single point of failures for a whole RAID group, it is a common practice for a RAID group to span disks from multiple shelf enclosures.

In this section, we will study the statistical property of storage subsystem failures both from a shelf perspective and from a RAID group perspective.



(a) Failure distribution in a shelf



(b) Failure distribution in a RAID group

Figure 9. Distribution of time between failures in about 155,000 shelves and 230,000 RAID groups during 44 months.

5.1 Time Between Failures

Figure 9 (a) and (b) show the empirical cumulative distribution function (CDF) of time between storage subsystem failures from a shelf and from a RAID group, respectively. To study the failure distribution from different disks in the same shelf/RAID group, we filtered out all duplicate failures. Since we only know when the failures are detected, instead of when the failures occur, the CDFs do not start from “zero” point. As all the storage subsystems studied in this paper send data verification requests to all disks hourly, as a proactive method to detect failures, we expect a short lag (up to an hour) between the occurrence and the detection of storage subsystem failures.

Finding (8): *Physical interconnect failures, protocol failures, and performance failures show much stronger temporal locality (“bursty” pattern) than disk failures.*

Implications: RAID-based resiliency mechanisms, which are designed for handling *disk failures*, might not be effective in handling all storage subsystem failure types.

As can be seen in Figure 9 (a), *overall storage subsystem failures* are very “bursty.” About 48% of *overall storage subsystem failures* arrive at the same shelf within 10,000 seconds of the previous failure. As expected, *physical interconnect failures* show the highest temporal locality, while even *protocol failures* and *performance failures* show strong temporal locality. None of these failure types follows distributions commonly used in failure theory, such as exponential distribution, Gamma distribution, or Weibull distribution.

On the other hand, *disk failures* show a much less “bursty” pattern, and the Gamma distribution provides a best fit for *disk failure*. For *disk failures*, we cannot reject the null hypothesis that *disk failures* follow the Gamma distribution with the Chi-Square-Test at the significance level of 0.05.

Finding (9): *Storage subsystem failures from a RAID group exhibit lower temporal locality (less “bursty” pattern) than failures from a shelf enclosure.*

Implications: Spanning RAID groups across multiple shelves is an effective way to reduce the probability for multiple storage subsystem failures to happen during a short period of time.

As we mentioned above, it is common to build a RAID group across multiple shelves, in order to prevent shelf from being a single point of failure. As we found out from the storage logs, a RAID group on average spans about 3 shelves.

Figure 9 (b) shows the CDF of time between failures from the same RAID group. Compared to Figure 9 (a), failures are less “bursty.” About 30% of failures arrive at the same RAID group within 10,000 seconds of the previous failure, lower than 48% for failures from the same shelf enclosure. For all failure types, the temporal locality is reduced. This observation supports the common practice of building a RAID group across multiple shelves and encourages storage system designers to distribute RAID groups more sparsely.

Finding (10): *Storage subsystem failures* of one RAID group still exhibit strong temporal locality.

Implications: We need resiliency mechanisms that can handle “bursty” failures.

However, Figure 9 (b) still shows strong temporal locality, since multiple shelves may share the same physical interconnect, and a network failure can still affect all disks in the RAID group.

We repeated this analysis using data broken down by system classes and shelf enclosure models. In all cases, similar patterns and trends were observed.

5.2 Correlations Between Failures

Our analysis of the correlation between failures is composed of two steps:

(1) **Derive the theoretical failure probability model based on the assumption that failures are independent.**

(2) **Evaluate the assumption by comparing the theoretical probability against empirical results.**

Next, we describe the statistical method we use for deriving the theoretical failure probability model.

5.2.1 Statistical Method

We denote the probability for a shelf enclosure (including all mounted disks) to experience one failure during time T as $P(1)$ and denote the probability for it to experience two failures during T as $P(2)$. Let $f(t)$ specify the failure probability at moment t .

Assume failures are independent, then we know that

$$P(1) = \int_0^T f(t)dt \quad (1)$$

$$\begin{aligned} P(2) &= \int_{t_2}^T \left(\int_0^{t_2} f(t_1)dt_1 \right) dt_2 \\ &= \frac{1}{2} (2 * \int_{t_2}^T \left(\int_0^{t_2} f(t_1)dt_1 \right) dt_2) \\ &= \frac{1}{2} \left(\int_0^T f(t)dt \right)^2 \end{aligned} \quad (2)$$

Therefore,

$$P(2) = \frac{1}{2} P(1)^2 \quad (3)$$

and more generally (the proof is skipped due to limited space),

$$P(N) = \frac{1}{N!} P(1)^N \quad (4)$$

We can derive the same formula for RAID group failure probability by replacing shelf enclosure with RAID group in the derivation above.

It is important to notice that the relation shown in equation 3 is a variation of a more common form:

$$P(A_1, A_2) = P(A_1) * P(A_2) \quad (5)$$

The main difference is that we do not care about the order of failures in equation 3.

In the next section, we will compare this theoretically derived model against the empirical results collected from storage logs.

5.2.2 Correlation Results

To evaluate the theoretical relation between $P(1)$ and $P(2)$ shown in equation 3, we first calculate *empirical* $P(1)$ and *empirical* $P(2)$ from storage logs. *Empirical* $P(1)$ is the percentage of shelves (RAID groups) that have experienced exactly one failure during time T (we set T as one year), and *empirical* $P(2)$ is the percentage of the ones that have experienced exactly two failures during time T . Only storage systems that have been in the field for one year or more are considered.

Finding (11): For each failure type, storage subsystem failures are not independent. After one failure, probability of additional failures (of the same type) is higher.

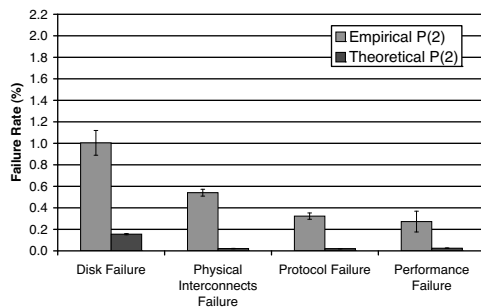
Implications: The probability of storage subsystem failures depends on factors shared by all disks in the same shelf enclosures (or RAID groups).

Figure 10 (a) shows the comparison between *empirical* $P(2)$ and *theoretical* $P(2)$, which is calculated based on *empirical* $P(1)$. As we can see in the figure, *empirical* $P(2)$ is higher than *theoretical* $P(2)$. More specifically, for *disk failure*, the observed *empirical* $P(2)$ is higher than *theoretical* $P(2)$ by a factor of 6. For other types of storage subsystem failures, the empirical probability is higher than the theoretical correspondences by a factor of 10-25. Furthermore, T-tests confirm that the *theoretical* $P(2)$ and the *empirical* $P(2)$ are statistically different with 99.5% confidence intervals.

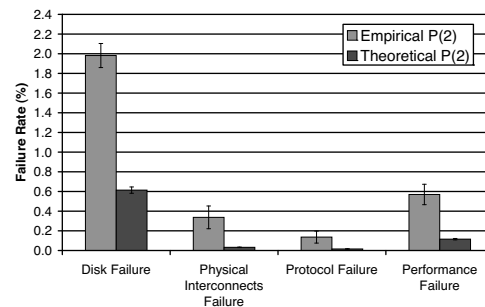
This is a strong indication that, when a shelf experiences a storage subsystem failure, the probability for it to have another storage subsystem failure increases. In another word, storage subsystem failures from the same shelves are not independent.

Figure 10 (b) shows the similar trend for the failures from the same RAID groups. Therefore, the same conclusion can be made for storage subsystem failures from the same RAID groups.

Although in Figure 10 we set T to be one year, the conclusion is general to different values of T . We have set



(a) Shelf enclosure failures



(b) RAID group failures

Figure 10. Comparison between theoretical model against empirical results. Theoretical $P(2)$ is calculated based on equation 3. The error bars show 99.5%+ confidence intervals.

T to 3 months, 6 months, and 2 years, and also grouped data based on other factors, such as system classes and shelf enclosure models. In all cases, similar correlations were observed.

5.2.3 Causes of Correlation

There are several reasons that can explain the correlation between each type of storage subsystem failures.

The *disk failure* probability depends on environmental factors, such as temperature [4]. Disks in the same shelf or the same RAID group are close to each other, sharing the same room temperature. Furthermore, disks in the same shelf are also sharing the cooling facility (*e.g.*, fans) provided by the shelf. When the machine room temperature is above or below the normal range, all disks in the same shelf and the same RAID group may experience a higher than normal failure probability. Similarly, when shelf cooling facility does not work properly, all disks in the same shelf may have higher probability to fail.

Most *physical interconnect* components, such as host adapters, cables, and FC terminators on the shelf, are shared by disks in the same shelf or in the same RAID group. When a *physical interconnect* component such as a host adapter experiences transient hardware errors, all disks in the same shelf or the same RAID group have a higher than normal probability of *physical interconnect failures*.

Similarly, drivers for disks in the same shelf or the same RAID group are usually updated around the same time. If a particular version is buggy or has compatibility issues, all disks will have a higher probability of *protocol failures*.

6 Related Work

Disk failure characteristic studies There are generally two categories of disk failure studies: vendor studies and user experience studies.

For example, Seagate and Quantum study long-term reliability characteristics through accelerated life tests of small populations and collecting statistics from their return unit databases [4, 21]. Based on such tests, they calculate the mean-time-to-failure (MTTF) and record it in a disk specification. For most of the disks, the specified MTTF is typically more than one million hours, equivalent to a lower than 1% annualized failure rate (AFR), which is slightly lower than what we observed (0.9-1.9%).

But vendor-specified MTTF is usually not what has been experienced by users. A study explained how disk manufactures and end customers can calculate MTTF in different ways [7].

Motivated by this observation, recently researchers have studied disk failures from a user's perspective by analyzing disk replacement logs collected in the field [14, 16]. Interestingly, they found disks are replaced much more frequently (2-4 times) than vendor-specified AFRs. More interestingly, [16] found that the time between disks replacements in the same machine room does not follow the exponential distribution and exhibits significant levels of correlation. This finding is consistent with what we find about the time between storage subsystem failures in the same shelf and the same RAID group, while we further found out that different failure types show different statistical properties.

Additionally, some researchers analyzed the characteristics of disk latent sector errors, which can potentially lead to complete *disk failures*, using the data from Net-

work Appliance AutoSupport Database as in [2]. Based on the same set of data, they further conducted a study on data corruption in [3]. They found enterprise class (e.g. FC) disks are more reliable than near-line (SATA) disks. Similarly, we discovered that FC disks have lower AFR (0.9%) compared to SATA disks (1.9%). However, we also observed that storage systems using FC disks are not necessarily more reliable than those using SATA disks, due to other component failures.

Some studies also look at the factors affecting disk failure rate, such as disk model, the number of disk heads, disk size, and environmental factors [2, 6, 14]. Similarly, in this paper, we looked at factors affecting storage failure rate, and found out that some factors strongly affecting storage failure rate have little impact on disk failure rate.

System component failure studies Unfortunately, there is little work published on analyzing the reliability of storage system components. Early work [17] presented a reliability analysis on disk array, and claimed that other system components such as power supplies, HBAs, cooling equipment, and cabling cannot be ignored when analyzing the reliability of a disk array. However, their study was not based on real-world data. Instead, they estimated reliability of disk array based on formula and datasheet-specified MTTF of each components, assuming component failures follow exponential distributions and failures are independent.

One of the very few empirical studies on storage system failures was presented in [20]. This paper presented an analysis of hardware failures in their prototype storage systems during 6 months. They found out that disks are among the most reliable components in the system, while SCSI components (physical interconnects in their prototypes) generated a considerable number of failures. These findings are consistent with our study. However, limited by the scale of the study, their failure sample size (limited to 16 storage systems and a few hundred failures) is too small to study important characteristics of failures such as failure distribution and failure correlations, nor to identify factors affecting storage system reliability. Another related empirical study looked at storage system outages based on 4,400 system-year records, and categorized the outages based on their root causes [11]. Although sharing the similar goal of categorizing failures, our study looks into the details of the storage subsystem failure, which is considered as one outage category in [11]. Furthermore, our study is based on data in a much larger scale (about 137,000 system-years).

Beyond storage systems, an analysis of Tandem systems found out that software errors are an increasing portion of failures reported by customers [10]. Similarly, we found that protocol stacks account for 5-10% of storage subsystem failures.

7 Conclusion

This paper presents a study of the real-world storage subsystem failures, examining the contribution of different failure types, the effect of some factors on failures, and the statistical properties of failures.

Our study is based on support logs collected from 39,000 commercially deployed storage systems, which contain about 1,800,000 disks mounted in about 155,000 shelf enclosures. The studied data cover a period of 44 months. The result of our study provides guidelines for designing more reliable storage systems and developing better resiliency mechanisms.

Although disks are the primary components of storage subsystems and disk failures contribute to 20-55% of storage subsystem failures, other components such as physical interconnects and protocol stacks also account for significant percentages (27-68% and 5-10%, respectively) of storage subsystem failures. The results clearly show that the rest of storage subsystem components cannot be ignored when designing a reliable storage system.

One way to improve storage system reliability is to select more reliable components. As data suggests, storage system reliability is highly dependent on both disk model and shelf enclosure model. We also found out that there might be a different “better” model for different storage systems, depending on other components used in the systems. Another way to improve reliability is to employ redundancy mechanisms to tolerate component failures. One such mechanism studied in the paper is multipathing, which can reduce AFR for storage systems by 30-40% when the number of paths is increased from one to two. Storage system designers should also think about using smaller shelves (fewer disks per shelf) but more shelves in storage systems, since data indicates that spanning a RAID group across multiple shelves can reduce the probability of “bursty” failures.

We also found out that storage subsystem failure and individual storage subsystem failure type exhibit strong self-correlations. In addition, these failures also exhibit “bursty” patterns. These results motivate a revisit to resiliency mechanisms such as RAID that assume independent failures.

Future work will compare the impact of different failure types and study how to design resiliency mechanisms targeting individual failure types, given that different failure types show different statistical properties. Another future direction is to design storage failure prediction algorithms based on component errors. We also want to extend this study to other components of storage systems beyond the storage subsystem.

8 Acknowledgments

We greatly appreciate our shepherd, Andrea Arpaci-Dusseau, for her invaluable feedback and precious time, and the anonymous reviewers for their insightful comments. We also wish to thank Rajesh Sundaram and Sandeep Shah for providing us with insights on storage failures. We are thankful to Frederick Ng, George Kong, Larry Lancaster, and Aziz Htite for offering help on understanding and gathering storage logs. We appreciate useful comments from members of the Advanced Development Group, including David Ford, Jiri Schindler, Dan Ellard, Keith Smith, James Lentini, Steve Byan, Sai Susarla, and Shankar Pasupathy. Finally, we would like to thank Lakshmi Bairavasundaram for his useful comments. This research has been funded by Network Appliance under the “Intelligent Log Mining” project at CS UIUC. Work of the first two authors was conducted in part as summer interns at Network Appliance.

References

- [1] B. Allen. Monitoring hard disks with smart. *Linux Journal*, 2004(117):9, 2004.
- [2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. *SIGMETRICS Perform. Eval. Rev.*, 35(1):289–300, 2007.
- [3] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA, Feb. 2008.
- [4] G. Cole. Estimating drive reliability in desktop computers and consumer electronics systems. Technical report, Seagate Technology Paper TP-338.1, 2000.
- [5] P. Corbett, B. English, A. Goel, T. Gnanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.
- [6] J. G. Elerath and S. Shah. Disk drive reliability case study: dependence upon head fly-height and quantity of heads. *Reliability and Maintainability Symposium*, pages 608–612, 2003.
- [7] J. G. Elerath and S. Shah. Server class disk drives: how reliable are they. *IEEE Reliability and Maintainability Symposium*, pages 151–156, 2004.
- [8] EMC Symmetrix DMX-4 Specification Sheet. http://www.emc.com/products/systems/symmetrix/symmetrix_DMX1000/pdf/DMX3000.pdf, July 2007.
- [9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003.
- [10] J. Gray. A census of tandem system availability between 1985 and 1990. In *Proceedings of the IEEE Transactions on Reliability*, 1990.
- [11] L. Lancaster and A. Rowe. Measuring real-world data availability. In *LISA '01: Proceedings of the 15th USENIX conference on system administration*, pages 93–100, Berkeley, CA, USA, 2001.
- [12] FAS6000 Series Technical Specifications. http://www.netapp.com/products/filer/fas6000_tech_specs.html.
- [13] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In H. Boral and P.-Å. Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 109–116. ACM Press, 1988.
- [14] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2007.
- [15] A. C. Rosander. *Elementary Principles of Statistics*. D. Van Nostrand Company, 1951.
- [16] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *FAST '07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2007.
- [17] M. Schulze, G. A. Gibson, R. H. Katz, and D. A. Patterson. How reliable is a RAID? In *COMPCON*, pages 118–123, 1989.
- [18] S. Shah and J. G. Elerath. Reliability analysis of disk drive failure mechanisms. In *IEEE Reliability and Maintainability Symposium*, pages 226–231, 2005.
- [19] Storage Networking Industry Association Dictionary. <http://www.snia.org/education/dictionary/>.
- [20] N. Talagala and D. Patterson. An analysis of error behaviour in a large storage system. Technical Report UCB/CSD-99-1042, EECS Department, University of California, Berkeley, Feb 1999.
- [21] J. Yang and F.-B. Sun. A comprehensive review of hard-disk drive reliability. In *Reliability and Maintainability Symposium*, pages 403–409, 1999.

Trademark Notice: NetApp, the Network Appliance logo are registered trademarks and Network Appliance is a trademark of Network Appliance, Inc. in the U.S. and other countries. All other brands or products are trademarks or registered trademarks of their respective holders and should be treated as such.

Parity Lost and Parity Regained

Andrew Krioukov*, Lakshmi N. Bairavasundaram*, Garth R. Goodson†, Kiran Srinivasan†,
Randy Thelen†, Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*

*University of Wisconsin-Madison

†Network Appliance, Inc.

{krioukov, laksh, dusseau, remzi}@cs.wisc.edu,

{goodson, skiran, rthelen}@netapp.com

Abstract

RAID storage systems protect data from storage errors, such as data corruption, using a set of one or more integrity techniques, such as checksums. The exact protection offered by certain techniques or a combination of techniques is sometimes unclear. We introduce and apply a formal method of analyzing the design of data protection strategies. Specifically, we use model checking to evaluate whether common protection techniques used in parity-based RAID systems are sufficient in light of the increasingly complex failure modes of modern disk drives. We evaluate the approaches taken by a number of real systems under single-error conditions, and find flaws in every scheme. In particular, we identify a parity pollution problem that spreads corrupt data (the result of a single error) across multiple disks, thus leading to data loss or corruption. We further identify which protection measures must be used to avoid such problems. Finally, we show how to combine real-world failure data with the results from the model checker to estimate the actual likelihood of data loss of different protection strategies.

1 Introduction

Data reliability and integrity is vital to storage systems. Performance problems can be tuned, tools can be added to cope with management issues, but data loss is seen as catastrophic. As Keeton *et al.* state, data unavailability may cost a company "... more than \$1 million/hour", but the price of data loss is "even higher" [23].

In well-designed, high-end systems, disk-related errors are still one of the main causes of potential trouble and thus must be carefully considered to avoid data loss [25]. Fortunately, with simple disk errors (*e.g.*, an entire disk failing in a fail-stop fashion), designing protection schemes to cope with disk errors is not overly challenging. For example, early systems successfully handle the failure of a single disk through the use of mirroring or parity-based redundancy schemes [6, 24, 29].

Although getting an implementation to work correctly may be challenging (often involving hundreds of thousands of lines of code [38]), one could feel confident that the design properly handles the expected errors.

Unfortunately, storage systems today are confronted with a much richer landscape of storage errors, thus considerably complicating the construction of correctly-designed protection strategies. For example, disks (and other storage subsystem components) are known to exhibit latent sector errors, corruption, lost writes, misdirected writes, and a number of other subtle problems during otherwise normal operation [2, 3, 17, 21, 30, 37]. Thus, a fully-formed protection strategy must consider these errors and protect data despite their occurrence.

A number of techniques have been developed over time to cope with errors such as these. For example, various forms of checksumming can be used to detect corruption [4, 35]; combined with redundancy (*e.g.*, mirrors or parity), checksumming enables both the detection of and recovery from certain classes of errors. However, given the broad range of techniques (including sector checksums, block checksums, parental checksums, write-verify operations, identity information, and disk scrubbing, to list a few), exactly which strategies protect against which errors is sometimes unclear; worse, combining different approaches in a single system may lead to unexpected gaps in data protection.

We propose a more formal approach based on model checking [20] to analyze the design of protection schemes in modern storage systems. We develop and apply a simple *model checker* to examine different data protection schemes. Within the system, one first implements a simple logical version of the protection strategy under test; the model checker then applies different sequences of read, write, and error events to exhaustively explore the state space of the system, either producing a chain of events that lead to data loss or a "proof" that the scheme works as desired.

We use the model checker to evaluate a number of dif-

ferent approaches found in real RAID systems, focusing on parity-based protection and single errors. We find holes in all of the schemes examined, where systems potentially exposes data to loss or returns corrupt data to the user. In data loss scenarios, the error is detected, but the data cannot be recovered, while in the rest, the error is not detected and therefore corrupt data is returned to the user. For example, we examine a combination of two techniques – block-level checksums (where checksums of the data block are stored within the same disk block as data and verified on every read) and write-verify (where data is read back immediately after it is written to disk and verified for correctness), and show that the scheme could still fail to detect certain error conditions, thus returning corrupt data to the user.

We discover one particularly interesting and general problem that we call *parity pollution*. In this situation, corrupt data in one block of a stripe spreads to other blocks through various parity calculations. We find a number of cases where parity pollution occurs, and show how pollution can lead to data loss. Specifically, we find that data scrubbing (which is used to reduce the chances of double disk failures) tends to be one of the main causes of parity pollution.

We construct a protection scheme to address all issues we discover including parity pollution. The scheme uses a version-mirroring technique in combination with block-level checksums and physical and logical identity information, leading to a system that is robust to a full and realistic range of storage errors.

With analyses of each scheme in hand, we also show how a system designer can combine real data of error probability with our model checker's results to arrive upon a final estimation of data loss probability. Doing so enables one to compare different protection approaches and determine which is best given the current environment. An interesting observation that emerges from the probability estimations is the trade-off between a higher probability detected data loss versus a lower probability of undetectable data corruption. For example, this trade-off is relevant when one decides between storing checksums in the data block itself versus storing them in a parent block. Another interesting observation is that data scrubbing actually increases the probability of data loss significantly under a single disk error.

The rest of the paper is structured as follows. Section 2 discusses background, while Section 3 describes our approach to model checking. Section 4 presents the results of using the model checker to deconstruct a variety of protection schemes; Section 5 presents the results of our probability analysis of each scheme combined with real-world failure data. Section 6 describes related work and Section 7 concludes.

2 Background

We provide some background first on a number of protection techniques found in real systems, and then on the types of storage errors one might expect to see in modern systems.

2.1 Protection Techniques

Protection techniques have evolved greatly over time. Early multiple disk systems focused almost solely on recovery from entire disk failures; detection was performed by the controller, and redundancy (*e.g.*, mirrors or parity) was used to reconstruct data on the failed disk [12].

Unfortunately, as disk drives became bigger, faster, and cheaper, new and interesting failure modes began to appear. For example, Network ApplianceTM recently added protection against “lost writes” [37], *i.e.*, write requests that appear to have been completed by the disk, but (for some reason) do not appear on the media. Many other systems do not (yet) have such protections, and the importance of such protection is difficult to gauge.

This anecdote serves to illustrate the organic nature of data protection. While it would be optimal to simply write down a set of assumptions about the fault model and then design a system to handle the expected errors, in practice such an approach is not practical. Disks (and other storage subsystem components) provide an ever-moving target; tomorrow's disk errors may not be present today. Worse, as new problems arise, they must be incorporated into existing schemes, rather than attacked from first principles. This aspect of data protection motivates the need for a formal and rigorous approach to help understand the exact protection offered by combinations of techniques.

Table 1 shows the protection schemes employed by a range of modern systems. Although the table may be incomplete (*e.g.*, a given system may use more than the protections we list, as we only list what is readily made public via published papers, web sites, and documentation), it hints at the breadth of approaches employed as well as the on-going development of protection techniques. We discuss each of these techniques in more detail in Section 4, where we use the model checker to determine their efficacy in guarding against storage errors.

2.2 Storage Errors

We now discuss the different types of storage errors. Many of these have been discussed in detail elsewhere [2, 3, 30, 37]. Here, we provide a brief overview and discuss their frequency of occurrence (if known).

- **Latent sector errors:** These errors occur when data cannot be reliably read from the disk drive medium.

System	RAID	Scrubbing	Sector checksums	Block checksums	Parent checksums	Write Verify	Physical Identity	Logical Identity	Version Mirroring	Other
Hardware RAID card (say, Adaptec TM 2200 S [1])	✓									
Linux software RAID [16, 28]	✓	✓								
Pilot [31]								✓		✓
Tandem NonStop® [4]	✓		✓				✓			
Dell TM Powervault TM [14]	✓	✓	✓							✓
Hitachi Thunder 9500 TM [18, 19]	✓		✓			✓				
NetApp® Data ONTAP® [37]	✓	✓		✓		✓	✓	✓		
ZFS [36] with RAID-4	✓	✓			✓					

Table 1: **Protections in Real Systems.** This table shows the known protections used in real-world systems. Some systems have other protections: Pilot uses a scavenger routine to recover metadata, and Powervault uses a 1-bit “write stamp” and a timestamp value to detect data-parity mismatches. Systems may use further protections (details not made public).

The disk drive returns an explicit error code to the system when a latent sector error is encountered.

- **Corruptions:** As the name indicates, these errors are said to occur when the data stored in a disk block is corrupted by an element of the storage stack.
- **Torn writes:** Disk drives may end up writing only a portion of the sectors in a given write request. Often, this occurs when the drive is power-cycled in the middle of processing the write request.
- **Lost writes:** In rare cases, buggy firmware components may return a success code to indicate completion of a write, but not perform the write in reality.
- **Misdirected writes:** In other rare cases, buggy firmware may write the data to the wrong disk or the wrong location within a disk. The effect of this error is two-fold: the original disk location does not receive the write it is supposed to receive (lost write), while the data in a different location is overwritten (with effects similar to corruption or lost write).

Latent sector errors affect about 19% of nearline and about 2% of enterprise class disks within 2 years of use [2]. Corruptions or torn writes affect on average around 0.6% of nearline and 0.06% of enterprise class disks within 17 months of use [3]. Lost or misdirected writes occur in about 0.04% of nearline and 0.007% of enterprise class disks within the first 17 months of use [3]. While the lost write numbers seem rather low, it is important to note that when a company sells a few million disks, at least one (and likely many more) customers could be affected by lost writes every year.

2.3 Error Outcomes

Depending on the protection techniques in place, storage systems errors may have one or more of the following outcomes:

- **Data recovery:** The scenario where the protection strategy detects the error, and uses parity to successfully recover data.
- **Data loss:** The scenario where the protection strategy detects the error, but is unable to successfully recover data. In this case, the storage system reports an error to the user.
- **Corrupt data:** The scenario where the protection strategy does not detect the error, and therefore returns corrupt data to the user.

3 Model Checking

We have developed a simple model checker to analyze the design of various data protection schemes. The goal of the model checker is to identify all execution sequences, consisting of user-level operations, protection operations, and disk errors, that can lead to either data loss or corrupt data being returned to the user. The model checker exhaustively evaluates all possible states of a single RAID stripe by taking into account the effects of all possible operations and disk errors for each state.

We have chosen to build our own model checker instead of using an existing one since it is easier to build a simple model checker that is highly specific to RAID data protection; for example, the model checker assumes that the data disks are inter-changeable, thereby reducing

the number of unique states. However, there is no fundamental reason why our analysis cannot be performed on a different model checker.

Models for the model checker are built on top of some basic primitives. A RAID stripe consists of N disk blocks where the contents of each disk block is defined by the model using primitive components consisting of user data entries and protections. Since both the choice of components and their on-disk layout affect the data reliability, the model must specify each block as a series of entries (corresponding to sectors within a block). Each entry can be atomically read or written.

The model checker assumes that the desired unit of consistency is one disk block. All protection schemes are evaluated with this assumption as a basis.

3.1 Model Checker Primitives

The model checker provides the following primitives for use by the protection scheme.

- **Disk operations:** The conventional operations disk read and disk write are provided. These operations are atomic for each entry (sector) and not over multiple entries that form a disk block.
- **Data protection:** The model checker and the model in conjunction implement various protection techniques. The model checker uses model-specified knowledge of the protections to evaluate different states. For example, the result of checksum verification is part of the system state that is maintained by the model checker. Protections like parity and checksums are modeled in such a way that “collisions” do not occur; we wish evaluate the spirit of the protection, not the choice of hash function.

The model defines operations such as user read and user write based on the model checker primitives. For instance, a user write that writes a part of the RAID stripe will be implemented by the model using disk read and disk write operations, parity calculation primitives, and protection checks.

3.2 Modeling Errors

The model checker injects exactly one error during the analysis of the protection scheme. The different types of storage errors discussed in Section 2.2 are supported. We now describe how the different errors are modeled.

- **Latent sector errors:** These errors are modeled as inaccessible data – an explicit error is returned when an attempt is made to read the disk block. Disk writes always succeed; it is assumed that if

a latent sector error occurs, the disk automatically *remaps* [2] the sectors.

- **Corruptions:** These errors are modeled as a change in value of a disk sector that produces a new value (*i.e.*, no collisions).
- **Lost writes:** These errors are modeled by not updating any of the sectors that form a disk block when a subsequent disk write is issued.
- **Torn writes:** These errors are modeled by updating only a portion of the sectors that form a disk block when a subsequent disk write is issued.
- **Misdirected writes:** These errors manifest in two ways: (i) they appear as a lost write for the block the write was intended to (the target), and (ii) it overwrites a different disk location (the victim). We assume that the target and victim are on different RAID stripes (otherwise, it would be a double error), and therefore can be modeled separately. Thus, we need to model only the victim, since the effects of a lost write on the target is an error we already study. A further assumption we make is that the data being written is block-aligned with the victim. Thus, a misdirected write is modeled by performing a write to a disk block (with valid entries) without an actual request from the model.

3.3 Model Checker States

A state according to the model checker is defined using the following sub-states: (a) the validity of each data item stored in the data disks as maintained by the model checker, (b) the results of performing each of the protection checks of the model, and (c) whether valid data and metadata items can be regenerated from parity for each of the data disks. The data disks are considered interchangeable; for example, data disk $D0$ with corrupt data is the same as data disk $D1$ with corrupt data as long as all other data and parity items are valid in both cases. As with any model checker, the previously explored states are remembered to avoid re-exploration.

The output of the model checker is a state machine that starts with the RAID stripe in the clean state and contains state transitions to each of the unique states discovered by the model checker. Table 2 contains a list of operations and errors that cause the state transitions.

4 Analysis

We now analyze various protection schemes using the model checker. We add protection techniques – RAID,

Operation	Description	Notation
User read	Read for any data disk	$R(X)$
User write	Write for any combination of disks in the stripe (the model performs any disk reads needed for parity calculation)	$W()$ is any write, $W_{ADD}()$ is write with additive parity, $W_{SUB}()$ is subtractive; Parameters: $X+$ is “data disk X plus others”, $!X$ is “other than data disk X”, full is “full stripe”
Scrub	Read all disks, verify protections, recompute parity from data, and compare with on-disk parity	S
Latent sector error	Disk read to a disk returns failure	$F_{LSE}(X)$, $F_{LSE}(P)$ for data disk X and parity disk respectively
Corruption	A new value is assigned to a sector	$F_{CORRUPT}(X)$, $F_{CORRUPT}(P)$ for data disk X and parity disk respectively
Lost write	Disk write issued is not performed, but success is reported	$F_{LOST}(X)$, $F_{LOST}(P)$ for data disk X and parity disk respectively
Torn write	Only the first sector of a disk write is written, but success is reported	$F_{TORN}(X)$, $F_{TORN}(P)$ for data disk X and parity disk respectively
Misdirected write	A disk block is overwritten with data following the same layout as the block, but not meant for it	$F_{MISDIR}(X)$, $F_{MISDIR}(P)$ for data disk X and parity disk respectively

Table 2: **Model Operations.** This table shows the different sources of state transitions: (a) operations that are performed on the model, and (b) the different errors that are injected.

data scrubbing, checksums, write-verify, identity, version mirroring – one by one, and evaluate each setup. We restrict our analysis to the protection offered by the different schemes against a single error. Indeed, we find that most schemes cannot recover from even a single error (given the proper failure scenario).

4.1 Bare-bones RAID

The simplest of protection schemes is the use of parity to recover from errors. This type of scheme is traditionally available through RAID hardware cards [1]. In this scheme, errors are typically detected based on error codes returned by the disk drive.

Figure 1 presents the model of bare-bones RAID, specified using the primitives provided by the model checker. In this model, a user read command simply calls a RAID-level read, which in turn issues a disk read for all disks. The disk read primitive returns the “data” successfully unless a latent sector error is encountered. On a latent sector error, the RAID read routine calls the reconstruct routine, which reads the rest of the disks, and recovers data through parity calculation. At the end of a user read, in place of returning data to the user, a validity check primitive is called. This model checker primitive verifies that the data is indeed valid; if it is not valid, then the model checker has found a hole in the protection scheme that returns corrupt data to the user.

When one or more data disks are written, parity is recalculated. Unless the entire stripe is written, parity calculation requires disk reads. In order to optimize

the number of disk reads, parity calculation may be performed in an additive or subtractive manner. In additive parity calculation, data disks other than the disks being written are read and the new parity is calculated as the XOR over the read blocks and the blocks being written. In subtractive parity calculation, the old data in the disks being written and the old parity are first read. Then, the new parity is the XOR of old data, old parity, and new data. Since parity calculation uses data on disk, it should verify the data read from disk. We shall see in the subsections that follow that the absence of this verification could violate data protection.

When the model checker is used to evaluate this model and only one disk error is injected, we obtain the state machine shown in Figure 2. Note that the state machine shows only those operations that result in state transitions (*i.e.*, self-loops are omitted). The model starts in the `clean` state and transitions to different states when errors occur. For example, a latent sector error to data disk X places the model in state `DiskX LSE`. The model transitions back to `clean` state, when one of the following occurs: (a) user read to data disk X *i.e.* $R(X)$, (b) user write to data disk X plus 0 or more other disks that in turn causes a disk read to data disk X for subtractive parity calculation ($W_{SUB}(X+)$), and (c) user write to any disks that result in additive parity calculation, thereby either causing data disk X to be read or data disk X to be overwritten ($W_{ADD}()$). Thus, we see that the model can recover from a latent sector error to data disks. We also see that the model can recover from a latent sector error to the parity disk as well.


```

UserRead(Disks[]) {
    data[] = RaidRead(Disks[]);
    if(raid read failed)
        Declare double failure and return;
    else
        CheckValid(Disks[], data[]);
}

RaidRead(Disks[]) {
    for(x = 0 to num(Disks[])) {
        data[x] = DiskRead(Disks[x]);
        if(disk read failed) { // LSE
            data[x] = Reconstruct(Disks[x]);
            if(reconstruct failed) { // another LSE
                return FAILURE;
            }
        }
    }
    return data[];
}

Reconstruct(BadDisk) {
    for(x = 0 to num(AllDisks[])) {
        if(Disks[x] is not BadDisk)
            data[x] = DiskRead(Disks[x]);
        else
            data[x] = DiskRead(ParityDisk);
        if(disk read fails) // LSE
            return FAILURE;
    }
    new_data = Parity(data[x]);
    DiskWrite(Disks[x], new_data);
    return new_data;
}

UserWrite(Disks[], data[]) {
    if(Additive parity cost is lower for num(Disks[])) {
        other_disk_data[] = RaidRead(AllDisks[] - Disks[]);
        if(raid read failed)
            Declare double failure and return;
        parity_data = Parity(data[] + other_disk_data[]);
    }
    else { // subtractive parity
        old_data[] = RaidRead(Disks[] + ParityDisk);
        if(raid read failed)
            Declare double failure and return;
        parity_data = Parity(data[] + old_data[]);
    }
    for(x = 0 to num(Disks[])) {
        DiskWrite(Disks[x], data[x]);
    }
    DiskWrite(ParityDisk, parity_data)
    return SUCCESS;
}

```

Figure 1: **Model of Bare-bones RAID.** The figure shows the model of bare-bones RAID specified using the primitives **DiskRead**, **DiskWrite**, **ParityCalc**, and **CheckValid** provided by the model checker. **CheckValid** is called when returning data to the user and the model checker verifies if the data is actually valid.

Let us now consider the state transitions that lead to corrupt data being returned to the user. We retain the names of states involved in these transitions for other data protection schemes as well, since the role they play is similar across schemes.

Any of the errors, lost write, torn write, misdirected write, or corruption to data disk X when in `clean` state, places the model in state `DiskX Error`. In this state, data disk X contains wrong data and the (correct) parity on the stripe is therefore inconsistent with the data disks. A user read to data disk X will now return corrupt data to the user (`Corrupt Data`), simply because there is no means of verifying that the data is valid. If a user write to disks other than data disk X triggers additive parity calculation ($W_{ADD}(!X)$), the corrupt data in data disk X is used for parity calculation, thereby corrupting the parity disk as well. In this scenario, both data disk X and the parity disk contain corrupt data, but they are consistent. We term this process of propagating incorrect data to the parity disk during additive parity calculation as *parity pollution* and it corresponds to the state `Polluted Parity`. Parity pollution does not impact the probability of data loss or corruption in this case since bare-bones RAID does not detect any form of corruption. However, as we shall see, parity pollution causes problems for many other protection schemes.

When in state `DiskX Error`, if a user write involving data disk X leads to subtractive parity calculation ($W_{SUB}(X+)$), the corrupt data in data disk X is used for the parity calculation. Therefore, the new parity generated is corrupt (and also inconsistent with the data disks). However, since data disk X is being written, data disk X is no longer corrupt. This state is named as `Parity Error` in the state machine. We see that the same state can be reached from `clean` state when an error occurs for the parity disk. This state does not lead to further data loss or corruption in the absence of a second error (if a second error is detected on one of the data disks, the corruption will be propagated to that disk as well). Thus, we see that, bare-bones RAID protects only against latent sector errors and not other errors.

4.2 Data Scrubbing

In this scheme, we add data scrubbing to the bare-bones RAID protection scheme. Data scrubbing is an extended version of disk media scrubbing [22, 33]. Data scrubs read all disk blocks that form the stripe and reconstruct the data if an error is detected. The scrub also recomputes the parity of the data blocks and compares it with the parity stored on the parity disk, thereby detecting any inconsistencies [3]. Thus, the scrubbing mechanism can convert the RAID recovery mechanism into an error detection technique. Note that if an inconsistency is detected,

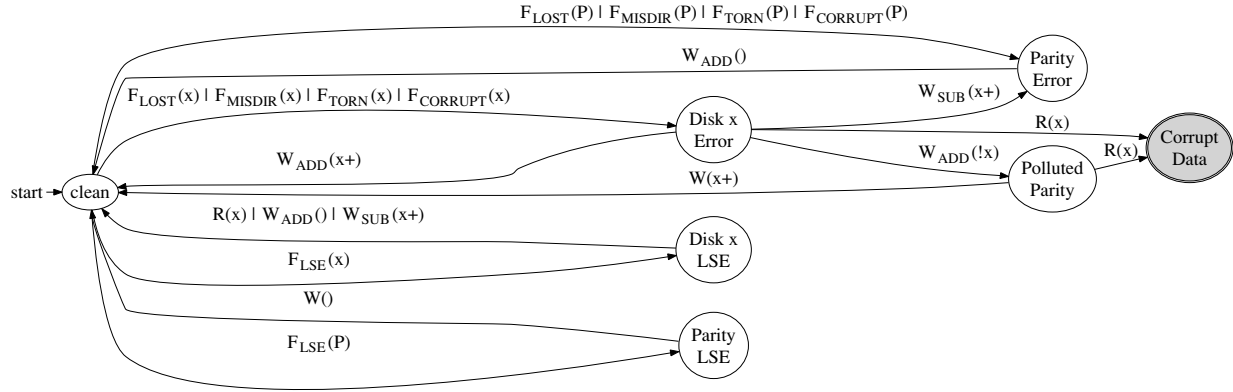


Figure 2: **State Machine for Bare-bones RAID.**

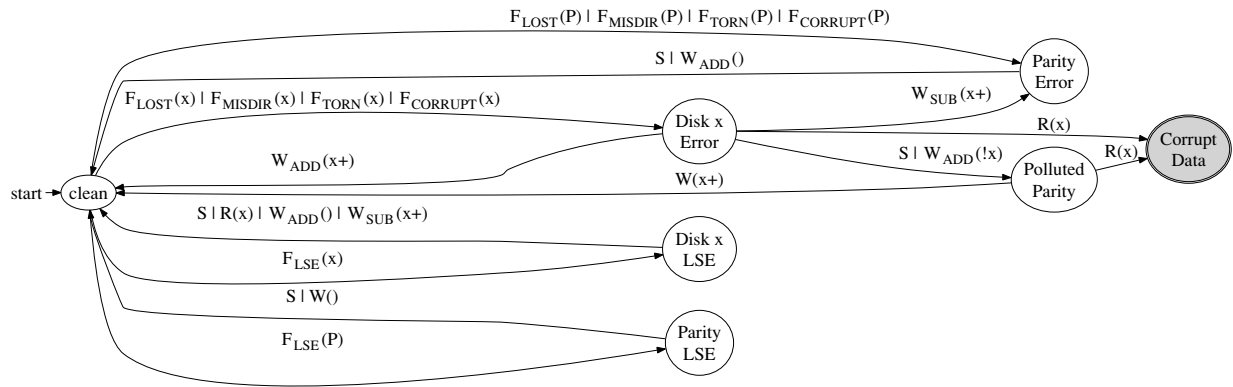


Figure 3: **RAID with scrubbing.**

bare-bones RAID does not offer a method to resolve it. The scrub should fix the inconsistency (by recomputing the contents of the parity disk) because inconsistent data and parity lead to data corruption if a second failure occurs and reconstruction is performed. In the rest of the section, when we refer to data scrubbing, we also imply that the scheme fixes parity inconsistencies.

When the model checker is used to examine this model and only one error is injected, we obtain the state machine shown in Figure 3. We see that the state machine is very similar to that of bare-bones RAID, except that some edges include S . One such edge is the transition from the state $\text{Disk}_X \text{ Error}$, where data in data disk X is wrong, to Polluted Parity , where both the data and parity are wrong, but consistently so. This transition during a scrub is easily explained – in $\text{Disk}_X \text{ Error}$, the scrub detects a mismatch between data and parity and updates the parity to match the data

moving the model to state Polluted Parity . We see that the addition of the scrub has not improved protection when only one error is injected; scrubs are intended to lower the chances of double failures, not of loss from single errors. In fact, we shall see later that the tendency of scrubs to pollute parity increases the chances of data loss when only one error occurs.

4.3 Checksums

Checksumming techniques have been used in numerous systems over the years to detect data corruption. Some systems store the checksum along with the data that it protects [4, 14, 37], while other systems store the checksum on the access path to the data [35, 36]. We will explore both alternatives. We also distinguish between the schemes that store per sector checksums [4, 14] and those that use per-block checksums [37].

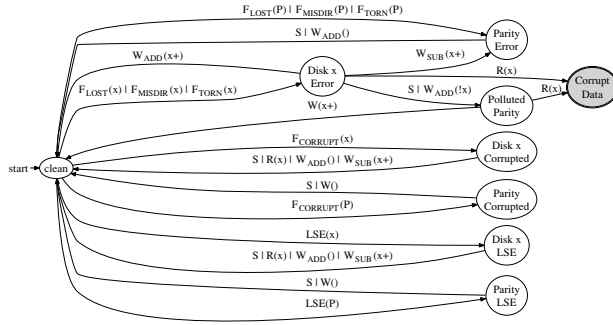


Figure 4: Sector checksums + RAID and scrubbing.

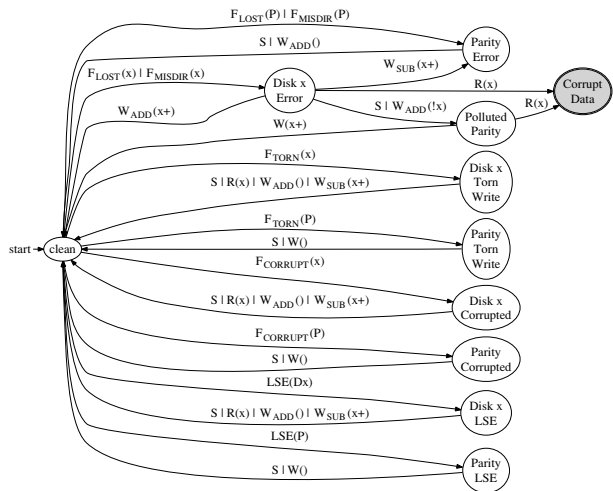


Figure 5: Block checksums + RAID and scrubbing.

Sector checksums: Figure 4 shows the state machine obtained for sector-level checksum protection. The obvious change from the previous state machines is the addition of two new states *Disk_X Corrupted* and *Parity Corrupted*. The model transitions to these states from the *clean* state when a corruption occurs to data disk *X* or the parity disk respectively. The use of sector checksums enables the detection of these corruptions whenever the corrupt block is read (including scrubs), thus initiating reconstruction and thereby returning the model to *clean* state. However, the use of sector checksums does not protect against torn writes, lost writes, and misdirected writes. For example, torn writes update a single sector, but not the rest of the block. The checksum for all sectors is therefore consistent with the data in that sector. Therefore, sector checksums do not detect these scenarios (*R(X)* from *Disk_X Error* leads to *Corrupt Data*).

Block checksums: The goal of block checksums is to ensure that a disk block is one consistent unit, unlike with sector checksums. Figure 5 shows the state machine obtained for block-level checksum protection. Again, the addition of new states that do not lead to *Corrupt Data* signifies an improvement in the protection. The new states added correspond to torn writes. Unlike sector-level protection, block-level protection can detect torn writes (detection denoted by transitions from states *Disk_X Torn* and *Parity Torn* to *clean*) in exactly the same manner as detecting corruption. However, we see that corrupt data could still be returned to the user. A lost write or a misdirected write transitions the model from the *clean* state to *Disk_X Error*. When a lost write occurs, the disk block retains data and checksum written on a previous occasion. The data and checksum are therefore consistent. Hence, the model does not detect that the data on disk *X* now returns corrupt data to the user. The scenario is similar for misdirected writes as well.

Parental checksums: A third option for checksumming is to store the checksum of the disk block in a parent block that is accessed first during user reads (e.g., an inode of a file is read before its data block). Parental checksums can thus be used to verify data during all user reads, but not for other operations. Figure 6 shows the state machine for this scheme. We notice many changes to the state machine as compared to block checksums. First, we see that the states successfully handled by block checksums (such as *F_TORN(X)*) do not exist. Instead, the transitions that led from *clean* to those states now place the model in *Disk_X Error*. Second, none of the states return corrupt data to the user. Instead, a new node called *Data Loss* has been added. This change signifies that the model detects a double failure and reports data loss. Third, the only transition to *Data Loss* is due to a read of data disk *X* when in the *Polluted Parity* state. Thus, parity pollution now leads to data loss. As before, the causes of parity pollution are data scrubs or additive parity calculations (transitions *S* or *W_ADD(!X)* lead from *Disk_X Error* to *Polluted Parity*). Figure 7 presents a pictorial view of the transitions from *clean* state to parity pollution and data loss. At the root of the problem is the fact that parental checksums can be verified only for user reads, not other disk reads. Any protection technique that does not co-operate with RAID, allows parity recalculation to use bad data, causing irreversible data loss.

Of the three checksums techniques evaluated, we find that block checksumming has the fewest number of transitions to data loss or corruption. Therefore, we use block checksums as the starting point for adding further protection techniques.

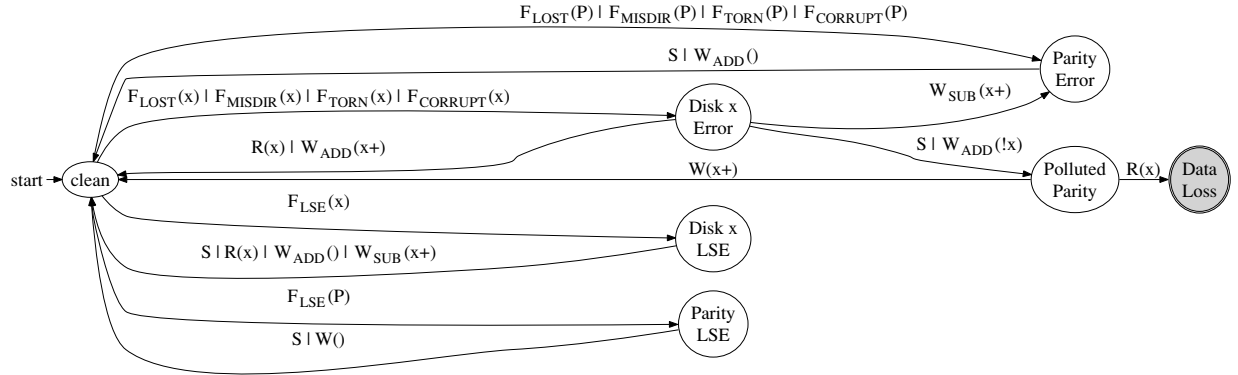


Figure 6: Parental checksums + RAID and scrubbing.

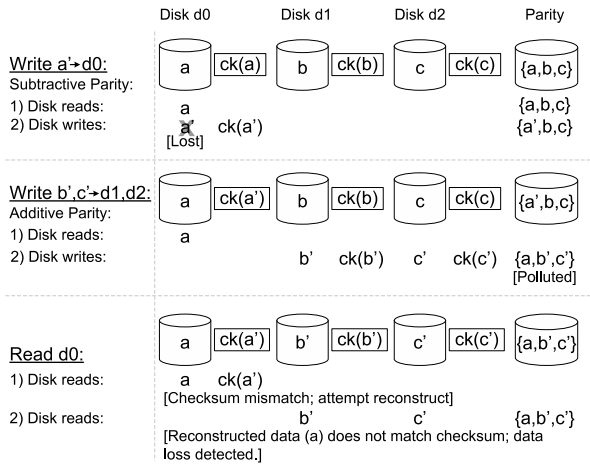


Figure 7: **Parity Pollution Sequence.** This figure shows a sequence of operations, along with intermediate RAID states, that lead to parity pollution and subsequent data loss. Each horizontal set of disks (Data disks d0, d1 and d2 and Parity disk) form the RAID stripe. The contents of the disk blocks are shown inside the disks. a, b etc. are data values, and $\{a, b\}$ denotes the parity of values a and b . The protection scheme used is parental checksums. Checksums are shown next to the corresponding data disks. At each RAID state, user read or write operations cause corresponding disk reads and writes, resulting in the next state. The first write to disk d0 is lost, while the checksum and parity are successfully updated. Next, a user write to disks d1 and d2 uses the bad data in disk d0 to calculate parity, thereby causing parity pollution. A subsequent user read to disk d0 detects a checksum mismatch, but recovery is not possible since parity is polluted.

4.4 Write-Verify

One primary problem with block checksums is that lost writes are not detected. Lost writes are particularly difficult to handle. If the checksum is stored along with the data and both are written as part of the same disk re-

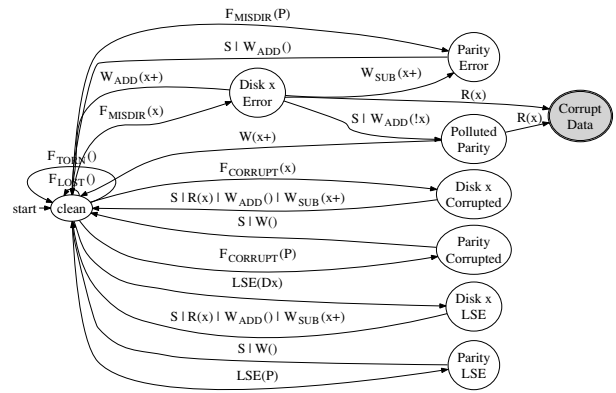


Figure 8: Write-verify + block checksums, RAID and scrubbing.

quest, they are both lost, leaving the old data and checksum intact and valid. On later reads to disk block, checksum verification compares the old data and old checksum which are consistent, thereby not detecting the lost write.

One simple method to fix this problem is to ensure that writes are not lost in the first place. Some storage systems perform write-verify [18, 37] (also called read-after-write verify) for this purpose. This technique reads the disk block back after it is written, and uses the data contents in memory to verify that the write has indeed completed without errors.

Figure 8 shows the state machine for write-verify with block checksums. Comparing this figure against Figure 5, we notice two differences: First, the states representing torn data or parity do not exist anymore. Second, the transitions $F_{TORN}(X)$, $F_{TORN}(P)$, $F_{LOST}(X)$, and $F_{LOST}(P)$ are now from `clean` to itself, instead of to

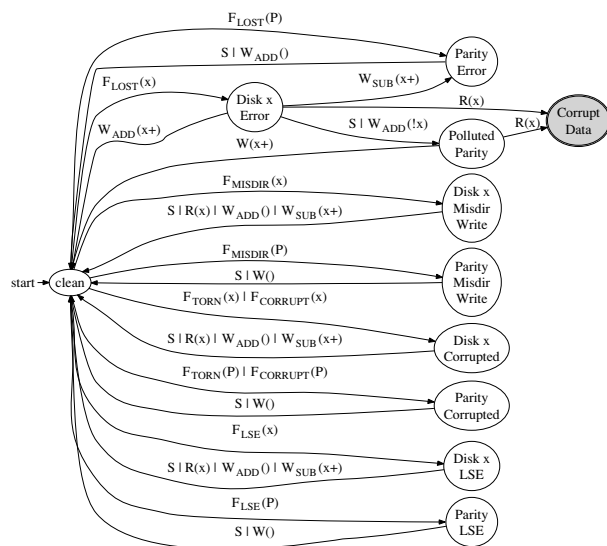


Figure 9: **Physical identity + RAID, scrubbing, and block checksums.**

other states (self-loops shown for readability). Write-verify detects lost writes and torn writes as and when they occur, keeping the RAID stripe in clean state.

Unfortunately, write-verify has two negatives. First, it does not protect against misdirected writes. When a misdirected write occurs, Write-verify would detect that the original target of the write suffered a lost write, and therefore simply reissue the write. However, the victim of the misdirected write is left consistent with consistent checksums but wrong data. A later user read to the victim thus returns corrupt data to the user. Second, although write-verify improves data protection, every disk write now incurs a disk read as well, possibly leading to a huge loss in performance.

4.5 Identity

A different approach that is used to solve the problem of lost or misdirected writes without the huge performance penalty of write-verify is the use of identity information.

Different forms of identifying data (also called self-describing data) can be stored along with data blocks. An identity may be in one of two forms: (a) physical identity, which typically consists of the disk number and the disk block (or sector) number to which the data is written [4], and (b) logical identity, which is typically an inode number and offset within the file [31, 37].

- **Physical identity:** Figure 9 shows the state machine obtained when physical identity information is used in combination with block checksums. Com-

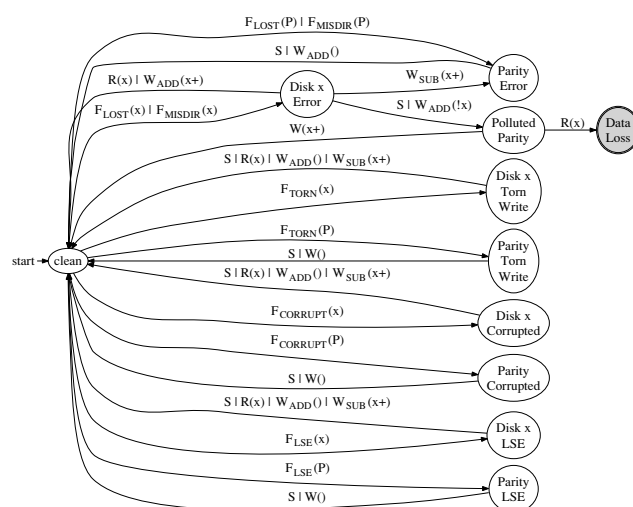


Figure 10: **Logical identity + RAID, scrubbing and block checksums.**

pared to previous state machines, we see that there are two new states corresponding to misdirected writes, Misdirected Data and Misdirected Parity. These states are detected by the model when the disk block is read for any reason (scrub, user read, or parity calculation) since even non-user operations like scrub can verify physical identity. Thus, physical identity is a step towards mitigating parity pollution. However, parity pollution still occurs in state transitions involving lost writes. If a lost write occurs, the disk block contains the old data, which would still have the correct physical block number. Therefore, physical identity cannot protect against lost writes, leading to corrupt data being returned to the user.

- **Logical identity:** The logical identity of disk blocks is defined by the block's parent and can therefore be verified only during user reads. Figure 10 shows the state machine obtained when logical identity protection is used in combination with block checksums. Unlike physical identity, misdirected writes do not cause new states to be created for logical identity. Both lost and misdirected writes place the model in the Disk_x Error state. At this point, parity pollution due to scrubs and user writes moves the system to the Polluted Parity state since logical identity can be verified only on user reads, thus causing data loss. Thus, logical identity works in similar fashion to parental checksums: (i) in both cases, there is a check that uses data from outside the block being protected, and (ii) in both cases, corrupt data is not returned to the user and instead, data loss is detected.

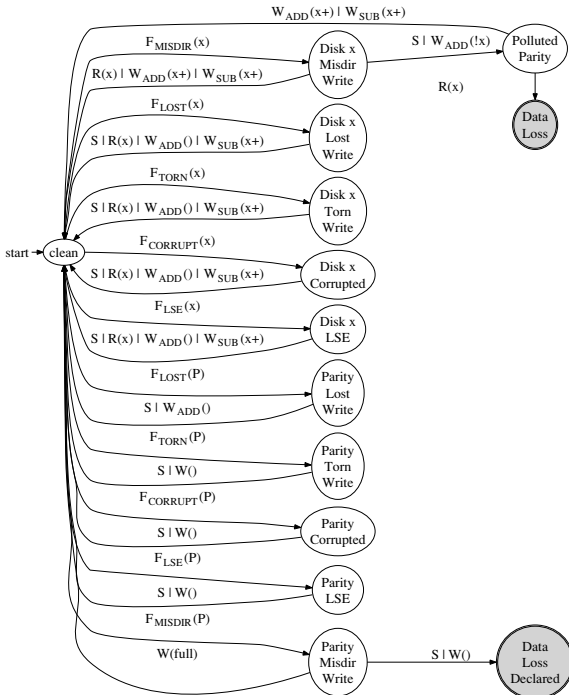


Figure 11: **Version mirroring + Logical identity, block checksums, RAID, and scrubbing.**

4.6 Version Mirroring

The use of identity information (both physical and logical) does not protect data from exactly one scenario – parity pollution after a lost write. Version mirroring can be used to detect lost writes during scrubs and parity calculation. Herein, each data block that belongs to the RAID stripe contains a version number. This version number is incremented with every write to the block. The parity block contains a list of version numbers of all of the data blocks that it protects. Whenever a data block is read, its version number is compared to the corresponding version number stored in the parity block. If a mismatch occurs, the newer block will have a higher version number, and can be used to reconstruct the other data block.

Note that when this approach is employed during user reads, each disk block read would now incur an additional read of the parity block. To avoid this performance penalty, version numbers can be used in conjunction with logical identity. Thus, logical identity is verified during file system reads, while version numbers are verified for parity re-calculation reads and disk scrubbing. This approach incurs an extra disk read of the parity block only when additive parity calculation is performed.

A primitive form of version mirroring has been used in real systems: Dell Powervault storage arrays [14] use a 1-bit version number called a “write stamp”. However,

since the length of the version number is restricted to 1-bit, it can only be used to *detect* a mismatch between data and parity (which we already can achieve through parity recompute and compare). It does not provide the power to identify the wrong data (which would enable recovery). This example illustrates that the bit-length of version numbers limits the number of errors that can be detected and recovered from.

Figure 11 shows the state machine obtained when version mirroring is added to logical identity protection. We find that there are now states corresponding to lost writes (Lost Data and Lost Parity) for which all transitions lead to clean. However, Data Loss could still occur, and in addition, Data Loss Declared could occur as well. The only error that causes state transitions to any of these nodes is a misdirected write.

A misdirected write to data disk X places the model in Misdirected Data. Now, an additive parity calculation that uses data disk X will compare the version number in data disk X against the one in the parity disk. The misdirected write could have written a disk block with a higher version number than the victim. Thus, the model trusts the wrong data disk X and pollutes parity. A subsequent read to data disk X uses logical identity to detect the error, but the parity has already been polluted.

A misdirected write to the parity disk causes problems as well. Interestingly, none of the protection schemes so far face this problem. The sequence of state transitions leading to Data Loss Declared occurs in following fashion. A misdirected write to the parity disk places new version numbers in the entire list of version numbers on the disk. When any data disk’s version number is compared against its corresponding version number on this list (during a write or scrub), if the parity’s (wrong) version numbers are higher, reconstruction is initiated. Reconstruction will detect that none of the version numbers of the data disks match the version numbers stored on the parity disk. In this scenario, a multi-disk error is detected and the model declares data loss. This state is different from Data Loss, since this scenario is a false positive while the other has actual data loss.

The occurrence of the Data Loss Declared state indicates that the policy used when multiple version numbers mismatch during reconstruction is faulty. It is indeed possible to have a policy that fixes parity instead of data on a multiple version number mismatch. The use of a model checker thus enables identification of policy faults as well.

We know from the previous subsection that physical identity protects against misdirected writes. Therefore, if physical identity is added to version mirroring and logical identity, we could potentially eliminate all problem nodes. Figure 12 shows the state machine generated for this protection scheme. We see that none of the state tran-

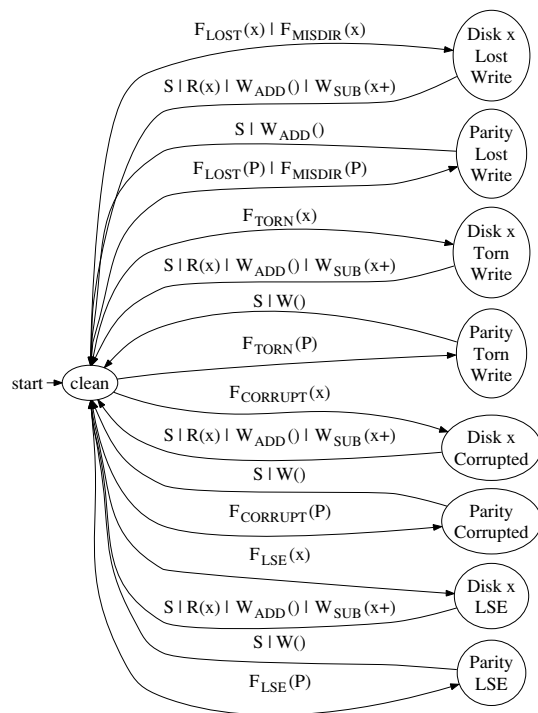


Figure 12: Version mirroring + Logical and physical identity, block checksums, RAID and scrubbing.

sitions lead to data loss or data corruption. The advantage of using physical identity is that the physical identity can be verified (detecting any misdirected write) before comparing version numbers. Thus, we have identified a scheme that eliminates data loss or corruption due to a realistic range of disk errors; the scheme includes version mirroring, physical and logical identity, block checksums, and RAID.

4.7 Discussion

The analysis of multiple schemes has helped identify the following key data protection issues.

- **Parity pollution:** We believe that any parity-based system that re-uses existing data to compute parity is potentially susceptible to data loss due to disk errors, in particular lost and misdirected writes. In the absence of techniques to perfectly verify the integrity of existing disk blocks used for recomputing the parity, disk scrubbing and partial-stripe writes can cause parity pollution, where the parity no longer reflects valid data.

In this context, it would be interesting to apply model checking to understand schemes with double parity [7, 13]. Another interesting scheme that could be analyzed is one with RAID-Z [8] protection (instead of RAID-4 or RAID-5), where only full-stripe writes are performed

and data is protected with parental checksums.

- **Parental protection:** Verifying the contents of a disk block against any value – either identity or checksum, written using a separate request and stored in a different disk location, is an excellent method to detect errors that are more difficult to handle. However, in the absence of techniques such as version mirroring, schemes that protect data by placing checksum or identity protections on the access path should use the same access path for disk scrubbing, parity calculation, and reconstructing data. Note that this approach could slow down these processes significantly, especially when the RAID is close to full space utilization.

- **Mirroring:** Mirroring of any piece of data, provides a distinct advantage: one can verify the correctness of data through comparison without interference from other data items (as in the case of parity). Version mirroring utilizes this advantage in conjunction with crucial knowledge about the items that are mirrored – the higher value is more recent.

- **Physical identity:** Physical identity, like block checksums, is extremely useful since it is knowledge available at the RAID-level. We see that this knowledge is important for perfect data protection.

- **Recovery-integrity co-design:** Finally, it is vital to integrate data integrity with RAID recovery, and do so by exhaustively exploring all possible scenarios that could occur when the protection techniques are composed.

Thus, a model checking approach is very useful in deconstructing the exact protection offered by a protection scheme, thereby also identifying important data protection issues. We believe that such an exhaustive approach would prove even more important in evaluating protections against double errors.

5 Probability of Loss or Corruption

One benefit of using a model checker is that we can assign probabilities to various state transitions in the state machine produced, and easily generate approximate probabilities for data loss or corruption. These probabilities help compare the different schemes quantitatively.

We use the data for nearline disks in Section 2 to derive per-year probabilities for the occurrence of the different errors. For instance, the probability of occurrence of F_{LSE} (a latent sector error) for one disk is 0.1. The data does not distinguish between corruption and torn writes; therefore, we assume an equal probability of occurrence of $F_{CORRUPT}$ and F_{TORN} (0.0022). We derive the probabilities for F_{LOST} and F_{MISDIR} based on the assumptions in Section 3.2 as 0.0003 and $1.88e-5$ respectively.

We also compute the probability for each operation to be the first to encounter the stripe with an existing er-

RAID	Scrub	Sector Checksum	Block Checksum	Parent Checksum	Write-Verify	Physical ID	Logical ID	Version Mirror	Chance of Data Loss
✓									0.602%
✓	✓								0.602%
✓	✓	✓							0.322%
✓	✓		✓						0.041%
✓	✓			✓					*0.486%
✓				✓					*0.153%
✓	✓		✓		✓				0.002%
✓	✓		✓			✓			0.038%
✓	✓		✓				✓		*0.033%
✓			✓				✓		*0.010%
✓	✓		✓			✓	✓		*0.031%
✓			✓			✓	✓		*0.010%
✓	✓		✓				✓	✓	*0.004%
✓			✓				✓	✓	*0.002%
✓	✓		✓			✓	✓	✓	0.000%

Table 3: Probability of Loss or Corruption. *The table provides an approximate probability of at least 1 data loss event and of corrupt data being returned to the user at least once, when each of the protection schemes is used for storing data. It is assumed that the storage system uses 4 data disks, and 1 parity disk. A (*) indicates that the data loss is detectable given the particular scheme (and hence can be turned into unavailability, depending on system implementation).*

ror. For this purpose, we utilize the distribution of how often different requests detect corruption in our study study [3]. The distribution is as follows. P(User read): 0.2, P(User write): 0.2, P(Scrub): 0.6. We assume that partial stripe writes of varying width are equally likely.

Note that while we attempt to use as realistic probability numbers as possible, the goal is not to provide precise data loss probabilities, but to illustrate the advantage of using a model checker, and discuss potential trade-offs between different protection schemes.

Table 3 provides approximate probabilities of data loss derived from the state machines produced by the model checker. We consider a 4 data disk, 1 parity disk RAID configuration for all of the protection schemes for calculating probabilities. This table enables simple comparisons of the different protection schemes. We can see that generally, enabling protections causes an expected decrease in the chance of data loss. The use of version mirroring with logical and physical identity, block checksums and RAID produces a scheme with a theoretical chance of data loss or corruption as 0. The data in the

table illustrates the following trade-offs between protection schemes:

Scrub vs. No scrub: Systems employ scrubbing to detect and fix errors and inconsistencies in order to reduce the chances of double failures. However, our analysis in the previous section shows that scrubs could potentially cause data loss due to parity pollution. The data in the table shows that it is indeed the case. In fact, since scrubs have a higher probability of encountering errors, the probability of data loss is significantly higher with scrubs than without. For example, using parental checksums with scrubs causes data loss with a probability 0.00486, while using parental checksums without scrubs causes data loss with a 3 times lesser probability 0.00153.

Data loss vs. Corrupt data: Comparing the different protection schemes, we see that some schemes cause data loss whereas others return corrupt data to the user. Interestingly, we also see that the probability of data loss is higher than the probability of corrupt data. For example, using parental checksums (with RAID and scrubbing) causes data loss with a probability 0.00486, while using block checksums causes corrupt data to be returned with an order of magnitude lesser probability 0.00041. Thus, while in general it is better to detect corruption and incur data loss than to return corrupt data, the answer may not be obvious when the probability of loss is much higher.

If the precise probability distributions of the underlying errors, and read, write, and scrub relative frequencies are known, techniques like Monte-Carlo simulation can be used to generate actual probability estimates that take multiple errors into consideration [15].

6 Related Work

Many research efforts have explored reliability modeling for RAID-based storage systems, right from when the case was made for RAID storage [29]. Most initial efforts focus on complete disk failures [10, 11, 26, 27]. For example, Burkhard and Menon [10] use Markov models to estimate the reliability provided by multiple check (parity) disks in a RAID group.

More recent research has explored the impact of partial disk failures, such as latent sector errors. Disk scrubbing [22] has been used for many years for proactive detection of latent errors, thus reducing the probability of double failures. Schwarz *et al.* use statistical models to analyze the fault tolerance provided by different options for disk scrubbing in archival storage systems [33]. El-erath and Pecht use Monte Carlo simulation to explore RAID reliability, considering different distributions for disk failures, latent errors, disk scrubbing, and time taken for RAID reconstruction [15]. Most of these research ef-

forts compute the reliability of RAID systems assuming that errors are detected and fixed when encountered (say through scrubbing), while we examine the design of the protections that provide such an assurance.

Sivathanu *et al.* provide a qualitative discussion of the assurances provided by various redundancy techniques [34]. We show that when multiple techniques are used in combination, a more exhaustive exploration of such assurances is essential.

Most related to our work is simultaneous research by Belluomini *et al.* [5]. They describe how undetected disk errors such as silent data corruptions and lost writes could potentially lead to a RAID returning corrupt data to the user. They explore a general solution space involving addition of an appendix with some extra information to each disk block or the parity block. In comparison, our effort is a detailed analysis of the exact protection offered by each type of extra information like block checksums, physical identity, etc.

Research efforts have also applied fault injection, instead of modeling, as a means to quantify the reliability and availability of RAID storage. Brown and Patterson [9] use benchmarks and fault injection to measure the availability of RAID.

Other research efforts that have leveraged model checking ideas to understand the reliability properties of actual operating system and storage system code [32, 39, 40]. For example, Yang *et al.* use model checking to identify bugs in file system code [40], and later they adapt model checking ideas to find bugs in many different file system and RAID implementations [39]. Model checking has also been used to study security protocols [32].

7 Conclusion

We have presented a formal approach to analyzing the design of data protection strategies. Whereas earlier designs were simple to verify by inspection (*e.g.*, a parity disk successfully adds protection against full-disk failure), modern systems employ a host of techniques, and their interactions are subtle.

With our approach, we have shown that a variety of approaches found in past and current systems are successful at detecting a variety of problems but that some interesting corner-case scenarios can lead to data loss or corruption. In particular, we found that the problem of parity pollution can propagate errors from a single (bad) block to other (previously good) blocks, and thus lead to a gap in protection in many schemes. The addition of version mirroring and proper identity information, in addition to standard checksums, parity, and scrubbing, leads to a solution where no single error should (by design) lead to data loss.

In the future, as protection evolves further to cope with the next generation of disk problems, we believe approaches such as ours will be critical. Although model checking implementations is clearly important [40], the first step in building any successful storage system should begin with a correctly-specified design.

Acknowledgments

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance.

We thank members of the RAID group at NetApp including Atul Goel, Tomislav Gracanac, Rajesh Sundaram, Jim Taylor, and Tom Theaker for their insightful comments on data protection schemes. David Ford, Stephen Harpster, Steve Kleiman, Brian Pawlowski and members of the Advanced Technology Group at NetApp provided excellent feedback on the work. Finally, we would like to thank our shepherd Darrell Long and the anonymous reviewers for their detailed comments that helped improve the paper significantly.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

- [1] Adaptec, Inc. Adaptec SCSI RAID 2200S. http://www.adaptec.com/en-US/support/raid/scsi_raid/ASR-2200S/, 2007.
- [2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, CA, June 2007.
- [3] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.
- [4] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.
- [5] W. Belluomini, V. Deenadhayalan, J. L. Hafner, and K. Rao. Undetected Disk Errors in RAID Arrays. *To appear in the IBM Journal of Research and Development*.
- [6] D. Bitton and J. Gray. Disk shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB 14)*, pages 331–338, Los Angeles, California, August 1988.
- [7] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*, pages 245–254, Chicago, Illinois, April 1994.

- [8] J. Bonwick. RAID-Z. http://blogs.sun.com/bonwick/entry/raid_z, Nov. 2005.
- [9] A. Brown and D. A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.
- [10] W. Burkhard and J. Menon. Disk Array Storage System Reliability. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 432–441, Toulouse, France, June 1993.
- [11] P. Chen and E. K. Lee. Striping in a RAID Level 5 Disk Array. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 136–145, Ottawa, Canada, May 1995.
- [12] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [13] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, California, April 2004.
- [14] M. H. Darden. Data Integrity: The Dell—EMC Distinction. http://www.dell.com/content/topics/global.aspx/power/en/ps2q02_darden?c=us&cs=555&l=en&s=biz, May 2002.
- [15] J. Elerath and M. Pecht. Enhanced Reliability Modeling of RAID Storage Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2007)*, Edinburgh, United Kingdom, June 2007.
- [16] Gentoo HOWTO. HOWTO Install on Software RAID. http://gentoo-wiki.com/HOWTO_Gentoo_Install_on_Software_RAID, Sept. 2007.
- [17] R. Green. EIDE Controller Flaws Version 24. <http://mindprod.com/jgloss/eideflaw.html>, February 2005.
- [18] Hitachi Data Systems. Hitachi Thunder 9500 V Series with Serial ATA: Revolutionizing Low-cost Archival Storage. www.hds.com/assets/pdf/wp_157_sata.pdf, May 2004.
- [19] Hitachi Data Systems. Data Security Solutions. <http://www.hds.com/solutions/storage-strategies/data-security/solutions.html>, Sept. 2007.
- [20] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [21] H. H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, September 1997.
- [22] H. H. Kari, H. Saikkonen, and F. Lombardi. Detection of Defective Media in Disks. In *The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, October 1993.
- [23] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.
- [24] M. Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.
- [25] L. Lancaster and A. Rowe. Measuring Real World Data Availability. In *Proceedings of the LISA 2001 15th Systems Administration Conference*, pages 93–100, San Diego, California, December 2001.
- [26] J. Menon and D. Mattson. Comparison of Sparing Alternatives for Disk Arrays. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 318–329, Gold Coast, Australia, May 1992.
- [27] C. U. Orji and J. A. Solworth. Doubly Distorted Mirrors. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington, DC, May 1993.
- [28] J. Ostergaard and E. Bueso. The Software-RAID HOWTO. http://tldp.org/HOWTO/html_single/Software-RAID-HOWTO/, June 2004.
- [29] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.
- [30] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.
- [31] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.
- [32] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model Checking An Entire Linux Distribution for Security Violations. In *Proceedings of the Annual Computer Security Applications Conference*, Tucson, Arizona, Dec. 2005.
- [33] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Volendam, Netherlands, October 2004.
- [34] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *The 1st International Workshop on Storage Security and Survivability (StorageSS '05)*, Fairfax County, Virginia, November 2005.
- [35] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.
- [36] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.
- [37] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/mat1/sample/0206tot_resiliency.html, February 2006.
- [38] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.
- [39] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.
- [40] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

Enhancing Storage System Availability on Multi-Core Architectures with Recovery-Conscious Scheduling

Sangeetha Seshadri*
Subashini Balachandran†

Lawrence Chiu†
Clem Dickey†

Cornel Constantinescu†
Ling Liu* Paul Muench†

*Georgia Institute of Technology
801 Atlantic Drive GA-30332
{sangeeta,lingliu}@cc.gatech.edu

†IBM Almaden Research Center
650 Harry Road CA-95120
{lchiu, cornel, sbalach, pmuench}@us.ibm.com

Abstract

In this paper we develop a recovery conscious framework for multi-core architectures and a suite of techniques for improving the resiliency and recovery efficiency of highly concurrent embedded storage software systems. Our techniques aim at providing continuous availability and performance during recovery while minimizing the time to recovery and the need for rearchitecting the system (legacy code). The main contributions of our recovery conscious framework include (1) a task-level recovery model, which consists of mechanisms for classifying storage tasks into recovery groups and dividing the overall system resources into recovery-oriented resource pools, and (2) the development of recovery-conscious scheduling, which enforces some serializability of failure-dependent tasks in order to reduce the ripple effect of software failure and improve the availability of the system. We present three alternative recovery-conscious scheduling algorithms; each represents one way to trade-off between recovery time and system performance. We have implemented and evaluated these recovery-conscious scheduling algorithms on a real industry-standard storage system. Our experimental evaluation results show that the proposed recovery conscious scheduling algorithms are non-intrusive and can significantly improve (throughput by 16.3% and response time by 22.9%) the performance of the system during failure recovery.

1 Introduction

Enterprise storage systems are the foundations of most data centers today and extremely high availability is expected as a basic requirement from these systems. With rapid and exponential growth of digital information and the increasing popularity of multi-core architectures, the demand for large scale storage systems of extremely high availability (moving close to 7 nines) continues to grow. On the other hand, embedded storage software systems

(controllers) are becoming much more complex and difficult to test especially given concurrent development and quality assurance processes.

With software failures and bugs becoming an accepted fact, focusing on recovery and reducing time to recovery has become essential in many modern storage systems today. In current system architectures, even with redundant controllers, most microcode failures trigger system-wide recovery [9, 10] causing the system to lose availability for at least a few seconds, and then wait for higher layers to redrive the operation. This unavailability is visible to customers as service outage and will only increase as the platform continues to grow using the legacy architecture.

In order to reduce the recovery time and more importantly scale the recovery process with system growth, it is essential to perform recovery at a fine-grained level: recovering only failed components and allowing the rest of the system to function uninterrupted. However, due to fuzzy component interfaces, complex dependencies and involved operational semantics of the system, implementing such fine-grained recovery is challenging. Therefore, firstly we must develop a mechanism to perform fine-grained recovery taking into consideration interactions between components and recovery semantics. Secondly, since localized recovery spans multiple dependent threads in reality, we must bound this localized recovery process in time and resource consumption in order to ensure that resources are available for other normally operating tasks even during recovery. Finally, since we are dealing with a large legacy architecture (> 2M loc), to ensure feasibility in terms of development time and cost we should minimize changes to the architecture.

In this paper we develop a recovery conscious framework for multi-core architectures and a suite of techniques for improving the resiliency and recovery efficiency of highly concurrent embedded storage software systems. Our techniques aim at providing

continuous availability and good performance even during a recovery process by bounding the time to recovery while minimizing the need for rearchitecting the system.

The main contributions of our recovery conscious framework include (1) a task-level recovery model, which consists of mechanisms for classifying storage tasks into recovery groups and dividing the overall system resources into recovery-oriented resource pools; and (2) the development of recovery-conscious scheduling, which enforces some serializability of failure-dependent tasks in order to reduce the ripple effect of software failures and improve the availability of the system. We present three alternative recovery-conscious scheduling algorithms, each representing one way to trade-off between recovery time and system performance.

We have implemented and evaluated these recovery-conscious scheduling algorithms on a real industry-standard storage system. Our experimental evaluation results show that the proposed recovery conscious scheduling algorithms are non-intrusive, involve minimal new code and can significantly improve performance during failure recovery thereby enhancing availability.

2 Problem Definition

In this section we motivate this research and illustrate the problem we address by considering the storage controllers of some representative storage system architecture. We focus on system recoverability from software failures. Storage controllers are embedded systems that add intelligence to storage and provide functionalities such as RAID, I/O routing, error detection and recovery. Failures in storage controllers are typically more complex and more expensive to recover if not handled appropriately. Although this section discusses specifically about embedded software failures in a storage controller, we believe that most of the concepts may be applicable to other highly concurrent system software too.

2.1 Motivation and Technical Challenges

Figure 1 gives a conceptual representation of a storage subsystem. This is a single storage subsystem node consisting of hosts, devices, a processor complex and the interconnects. In practice, storage systems may be composed of one or more such nodes in order to avoid single-points-of-failure. The processor complex provides the management functionalities for the storage subsystem. The system memory available within the processor complex serves as program memory and may also serve as the data cache. The memory is accessible to all the processors within the complex and holds the job queues through which

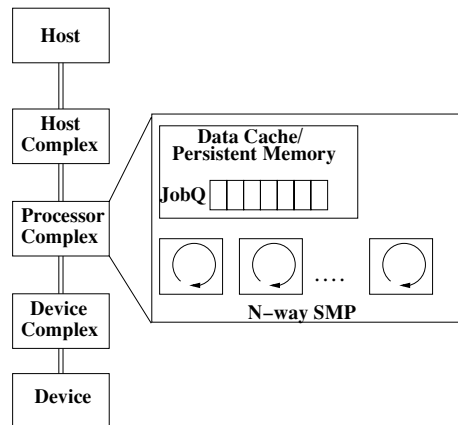


Figure 1: Storage Subsystem Architecture

functional components dispatch work. As shown in Figure 1, this processor complex has a single job queue and is an N-way SMP node. Any of the N processors may execute the jobs available in the queue. Some storage systems may have more than one job queue (e.g. multiple priority queues).

The storage controller software typically consists of a number of interacting components each of which performs work through a large number of asynchronous, short-running threads ($\sim \mu\text{secs}$). We refer to each of these threads as a ‘task’. Examples of components include SCSI command processor, cache manager and device manager. Tasks (e.g., processing a SCSI command, reading data into cache memory, discarding data from cache etc.) are enqueued onto the job queues by the components and then dispatched to run on one of the many available processors each of which runs an independent scheduler. Tasks interact both through shared data-structures in memory as well as through message passing.

With this architecture, when one thread encounters an exception that causes the system to enter an unknown or incorrect state, the common way to return the system to an acceptable, functional state is by restarting and reinitializing the entire system. Since the system state may either be lost, or cannot be trusted to be consistent, some higher layer must now redrive operations after the system has performed basic consistency checks of non-volatile metadata and data. While the system reinitializes and waits for the operations to be redriven by a host, access to the system is lost contributing to the downtime. This recovery process is widely recognized as a barrier to achieving high(er) availability. Moreover, as the system scales to larger number of cores and as the size of the in-memory structures increase, such system-wide recovery will no longer scale.

The necessity to embark on system-wide recovery to deal with software failures is mainly due to the complex interactions between the tasks which may

belong to different components. Due to the high volume of tasks (more than 20 million/minute in a typical workload), their short-running nature and the involved semantics of each task, it becomes infeasible to maintain logs or perform database-style recovery actions in the presence of software failures. Often such software failures need to be explicitly handled by the developer. However, the number of scenarios are so large, especially in embedded systems, that the programmer cannot realistically anticipate every possible failure. Also, an individual developer may only be aware of the clean-up routines for the limited scope being handled by them. This knowledge is insufficient to recover the entire system from failures, given that often interactions among tasks and execution paths are determined dynamically.

Many software systems, especially legacy systems, do not satisfy the conditions outlined as essential for micro-rebootable software [3]. For instance, even though the storage software may be reasonably modular, component boundaries, if they exist, are very loosely defined. In addition, the scenario where components are stateful and interact with other components through globally shared structures (data-structures, metadata), often leads to components modifying each other's state irreversibly. Moreover, resources such as hardware and software locks, devices and metadata are shared across components. Under these circumstances, the scope of a recovery action is not limited to a single component.

The discussion above highlights some key problems that need to be addressed in order to improve system availability and provide scalable recovery from software failures. Concretely, we must answer the following questions:

- How do we implement fine-grained recovery in a highly concurrent storage system?
- Can we identify recovery dependencies across tasks and construct efficient recovery scopes?
- How do we ensure availability of the system during a recovery process? What are important factors that will impact the recovery efficiency?

In addition to maintaining system performance while reducing the time to recovery, another key challenge in developing a scalable solution is to ensure that the recovery-conscious framework is non-intrusive and thus minimize re-architecting of the legacy application code. We will describe our solution to the first two problems — how to implement localized recovery and how to discover efficient recovery scopes in Section 3. We will dedicate Section 4 to address the third problem: how do we bound the recovery process and ensure system availability even during localized recovery?

2.2 Taxonomy of Failures

Studies classify software faults as both permanent and transient. Gray [6] classifies software faults into *Bohrbugs* and *Heisenbugs*. Bohrbugs are essentially deterministic bugs that may be caused due to permanent design failures. Such bugs are usually easily identified during the testing phases and are weeded out early in the software life cycle. On the other hand, 'heisenbugs' which are transient or intermittent faults that occur only under certain conditions are not easily identifiable and may not even be reproducible. Such faults are often due to reasons such as the system entering an unexpected state, insufficient exception handling, boundary conditions, timing/concurrency issues or due to other external factors. Many studies have shown that most software failures occurring in production systems are due to transient faults that disappear when the system is restarted [6, 3, 15].

Our work is targeted at dealing with such transient failures in a storage software system and in particular the embedded storage controller's microcode. Below, we provide a classification of transient failures which we intend to deal with through localized recovery.

In complex systems, often code paths are dynamic and input parameters are determined at runtime. As a result many faults are not caught at compile time. On pure functions, faults may be classified as:

- **Domain errors:** are caused by bad input arguments, such as a divide by zero error or when each individual input is correct, but the combination is wrong (e.g. negative number raised to a non-integral power in a real arithmetic system).
- **Range errors:** are caused when input arguments are correct, but the result cannot be computed (such as a result which would cause an overflow).

With actions based on system state there are additional complexities. For example, a configuration issue that appeared early in the installation process may have been fixed by trying various combinations of actions that were not correctly undone. As a result the system finds itself in an unknown state which manifests as a failure after some period of normal operation. Such errors are very difficult to trace, and although transient may continue to appear every so often. We classify such system state based errors as:

- **State error:** where the input arguments are wrong for the current state of the object.
- **Internal logic error:** where the system has unexpectedly entered an incorrect or unknown state. Such an error often triggers further state

errors.

Each of the above error types can lead to transient failures. Some of the transient failures can be fixed through appropriate recovery actions which may range from dropping the current request to retrying the operation or performing a set of actions that take the system to a known consistent state. For example, some of such transient faults that occur in storage controller code are:

- **Unsolicited response from adapter:** An adapter (a hardware component not controlled by our microcode) sends a response to a message which we did not send - or do not remember sending. This is an example of a state error.
- **Incorrect Linear Redundancy Code (LRC):** A control block has the wrong LRC check bytes, for instance, due to an undetected memory error; an example of an internal logic error.
- **Queue full:** An adapter refuses to accept more work due to a queue full condition; an example of both an internal logic error and state error.

In addition, there are other error scenarios such as violation of a storage system or application service level agreements. The ‘time-out’ conditions are also very common in large scale embedded storage systems. While the legacy system grows along multiple dimensions, the growth is not proportional along all dimensions. As a result hard-coded constant time-out values distributed in the code base often create unexpected artificial violations.

2.3 Recovery Models

Intuitively we can see that localized recovery may be possible for many of the failure scenarios outlined above, and thus system-wide software reboots can be avoided. Sometimes even for situations of resolving deadlock or livelock, it may be sufficient if a minimal subset of tasks or components of the system undergo restarts (e.g., deadlock resolution in transactional databases [7]). Of course there are scenarios, such as severe memory corruption, where the only high-confidence way of repairing the fault is to perform system-wide clean-up.

In production environments, techniques for fault-tolerance, i.e., coping with the existence and manifestation of software faults can be classified into two primary categories with respect to the fault repairing methods: (1) those that provide *fault treatment*, such as restarts of the software, rebooting of the system and utilizing process pair redundancy; and (2) those that provide *error recovery*, such as checkpointing and log-based recovery. Alternatively, one can categorize the recovery models based on the granularity of the recovery scopes. All the above-

mentioned techniques could be applied to any recovery scope. In our context, we consider the following three types of recovery scopes:

- **System level:** Performing fault treatment at this level has proven to be an effective high-confidence way of recovering the system from transient faults [2], but has a high cost in terms of recovery time and the resulting system downtime. On the other hand performing error recovery at the system level through checkpointing and recovery can be prohibitively expensive for systems with very high volumes of workload and complex semantics.
- **Component level:** Both fault treatment and error recovery are more scalable and cost effective at this granularity. For fault treatment, the main challenge is identifying these ‘component boundaries’ especially in systems that do not have well defined interfaces. Again, the difficult hurdle to performing checkpoint/log-based error recovery at this level is understanding the semantics of operations.
- **Task level:** At this very fine-grained level, the issue of operational semantics still remains. However, performing fault treatment at this level is efficient both in terms of cost and system availability.

The main advantage of performing error recovery or fault-treatment at the task-level as compared to the component-level, is that it allows us to accommodate cross-component interactions and define ‘recovery boundaries’ in place of ‘component boundaries’. Our goal is to handle most of the failures and exceptions through task-level (localized) recovery, and avoid resorting to system-wide recovery unless it is absolutely necessary.

3 Task-level Recovery Framework

Transactional recovery in relational DBMSs is a success story of fine-grained error recovery, where the set of operations, their corresponding recovery actions and their recovery scopes are well-defined in the context of database transactions. However, this is not the case in many legacy storage systems. For example, consider the embedded storage controller in which tasks executed by the system are involved in more complex operational semantics, such as dynamic execution paths and complex interactions with other tasks. Under these circumstances, in order to implement task-level recovery, we have to deal with both the semantics of recovery and the identification of recovery scopes.

Recovery from a software failure involves choosing an appropriate strategy to treat/recover from the failure. The choice of recovery strategy depends

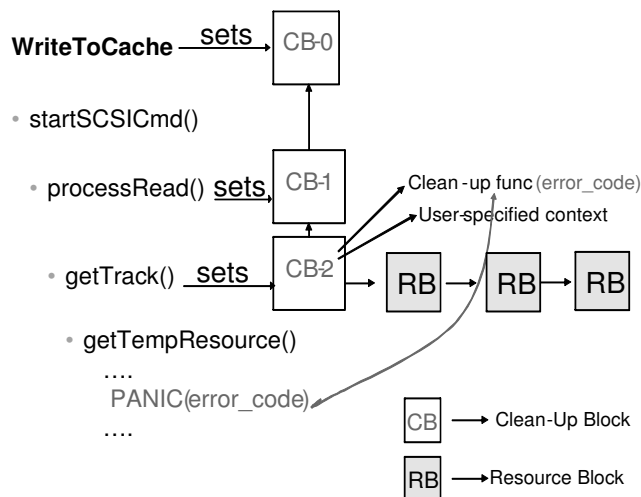


Figure 2: Framework for Task Level Recovery

on the nature of the task, the context of the failure, and the type of failure. For example, within a single system, the recovery strategy could range from continuing the operations (ignoring the error), retrying the operation (fault treatment using environmental diversity) or propagating the fault to a higher layer. In general, with every failure context and type, we could associate a recovery action. In addition, to ensure that the system will return to a consistent state, we must also avoid deadlock or resource hold-up situations by relinquishing resources such as hardware or software locks, devices or data sets that are in the possession of the task.

Bearing these design principles in mind, we develop a two-tier framework for performing task-level recovery through a set of non-intrusive recovery strategies. The top tier provides the capabilities of defining the recovery scope at task level through a careful combination of both the programmer's specification at much coarser granularity and the system-determined recovery scope at finer-granularity. The bottom tier provides the recovery-conscious scheduling algorithms that balance the performance and the recovery efficiency.

In this framework, we refer to the context of a failure as a **recovery point** and provide mechanisms for developers to define **clean-up blocks** which are recovery procedures and re-drive strategies. A clean-up block is associated with a recovery point and encapsulates failure codes, the associated recovery actions, and resource information. The specification of the actual recovery actions in each of the clean-up blocks is left to the developers due to their task-specific semantics.

In our implementation, the recovery-conscious scheduler alone was implemented in approximately 1000 lines of code. A naive coding and the design effort for task level recovery would be directly pro-

portional to the number of "panics" or failures in the code that are intended to be handled using our framework. In general, the coding effort for a single recovery action is small and is estimated to be around a few tens of lines of code (using semicolons as the definition of lines of code) per recovery action on average. Note that, the clean-up block does not involve any logging or complex book-keeping and is intended to be light-weight. A more efficient handling of clean-up blocks would involve classifying common error/failure situations and then addressing the handling of the errors in a hierarchical fashion. For example, recoveries may be nested and we could re-throw an error and recover with the next higher clean-up block defined in the stack. This would involve design effort towards the classification of error codes into classes and sub-classes and identification of common error handling situations. Finally, if we are unable to address an error using our framework, existing error handling mechanisms would be used as default. The point of recovery in the stack may be determined by factors such as access to data structures and possibilities of recovery strategies such as retrying, termination or ignoring the error.

Figure 2 shows a schematic diagram of the recovery framework using the call stack of a single task. As the task moves through its execution path, it passes through multiple recovery points and accumulates clean-up blocks. When the task leaves a context, the clean-up actions associated with the context go out of scope. On the other hand, nesting of contexts results in the nesting of the corresponding clean-up blocks and the framework keeps track of necessary clean-up blocks.

The clean-up blocks are gathered and carried along during task execution but are not invoked unless a failure occurs. Resource information can also be gathered passively. Such a framework allows a choice of recovery strategy based on task requirements and requires minimal rearchitecting of the system.

Example : We describe the selection of recovery strategy and design of clean-up blocks using an example from our storage controller implementation. Consider the error described in Figure 2 which depicts relevant portions of the call stack. The failure situation described in this example is similar to the commonly used 'assert' programming construct. The error is encountered when a task has run out of a temporary cache data structure known as a 'control block' which is not expected to occur normally and hence results in a 'panic'.

In this particular situation, ignoring the error is not a possible recovery strategy since the task would be unable to complete until a control block is available. One possible strategy is to search the list of

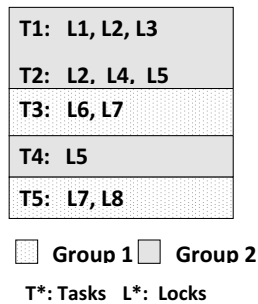


Figure 3: Implicit recovery scopes

control blocks to identify any instances that are not currently in use, but have not been freed correctly (for example, due to incorrect flags). If any such instances exist, they could be made available to the stalled task. An alternative strategy would be to retry the operation beginning at the ‘WriteToCache’ routine at a later time in order to work around concurrency issues. Retrying the operation may involve rolling back the resource and state setup along this call path to their original state. Resource blocks are used to carry the information required to successfully execute this strategy. Finally, in the case of less critical tasks, aborting the task may also be an option. Alternatively, consider a situation where an error is encountered due to a component releasing access to a track to which it did not have access in the first place. The error was caused due to a mismatch in the number of active users as perceived by the component. In this case, a possible recovery strategy would be to correctly set the count for the number of active users and proceed with the execution, effectively ignoring the error.

Note that, it is important we ensure that the interfaces with the recovery code and the recovery code itself are reliable. The most important issue in the tier one design is how to adequately identify the recovery scopes or boundaries, and how to concretely determine what are the set of tasks that need to undergo recovery upon a failure?

3.1 Identifying fine-grained recovery scopes

Tasks interact with each other in complex ways. When a single task encounters an exception, more than one task may need to initiate recovery procedures in order to avoid deadlocks and return the system to a consistent state. In order to identify the necessary and yet sufficient scope of a recovery action, we need to characterize dependencies between tasks.

Dependencies between tasks may be explicit as in the case of functional dependencies or implicit such as those arising from shared access to stateful structures (e.g., data structures, metadata) or

devices. For example, tasks belonging to the same user request may be identified as having the same recovery scope. Likewise, identical tasks belonging to the same functional component may also be marked with the same recovery scope. Explicit dependencies can be specified by the programmer.

However, explicit dependencies specified by the programmer may be very coarse. For example, an ‘adapter queue full’ error should only affect tasks attempting to write to that adapter and should not initiate recovery across the component. Likewise, some dependencies may have been overlooked due to their dynamic nature and the immense complexity of the system. Therefore one way to refine explicit dependencies is to identify implicit dependencies continuously and utilize them to refine the developer-defined recovery scopes over time. For example, one approach to identifying implicit dependencies at runtime is by observing patterns of hardware and software lock acquisitions. We can group the tasks that share locks into the same recovery scope, since sharing locks typically implies that they have shared access to a stateful global object. Figure 3 illustrates this approach through an example. It shows five tasks and their respective lock acquisition patterns. Tasks T1, T2 and T4 are accessing overlapping sets of locks during their execution and thus are grouped into one recovery scope. Similarly, tasks T3 and T5 are grouped into another recovery scope. Clearly, this approach further refines the developer-specified recovery scope at task level into smaller recovery scopes based on runtime characterization of the dependencies with respect to lock acquisition.

Due to the space limit of this paper, we will omit the detailed development of recovery scope refinement mechanisms. In the remaining part of this paper, we assume that tasks are organized into disjoint recovery scopes refined based on implicit dependencies identified dynamically at runtime. In addition to recovery scopes, we argue that recovery-conscious resource management can be critical for improving system availability during localized recovery.

3.2 Ensuring resource availability

Multi-core processors are delivering huge system-level benefits to embedded applications. Both SMP-based and multi-core systems are very popular in this segment. With the number of processing cores increasing continuously, we argue that the storage software needs to scale both in terms of performance and recoverability to take advantage of the system resources.

An important goal for providing fine-grained recovery (task or component level) is to improve recoverability and make efficient use of resources on the multi-core architectures. This ensures that re-

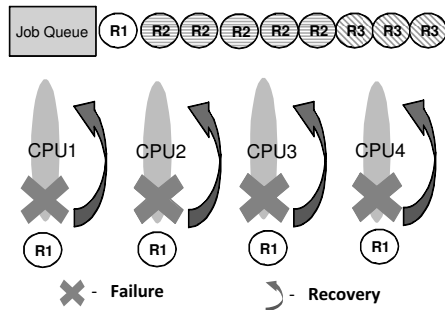


Figure 4: Current Scheduler

sources are available for normal system operation in spite of some localized recovery being underway and that the recovery process is bounded both in time and in resource consumption. Without careful design, it is possible that more dependent tasks are dispatched before a recovery process can complete, resulting in an expansion of the recovery scope or an inconsistent system state. This problem is aggravated by the fact that recovery takes orders of magnitude longer (ranging from milliseconds to seconds) compared to normal operation ($\sim \mu$ secs). Also a dangerous situation may arise where it is possible that many or all of the threads that are concurrently executing are dependent, especially since tasks often arrive in batches. Then the recovery process could consume all system resources essentially stalling the entire system.

Ideally we would like to “fence” the failed and recovering tasks until the recovery is complete. In order to do so we must control the number of dependent tasks that are scheduled concurrently, both during normal operation and during recovery. In the next section we discuss how to design a recovery conscious scheduler that can control how many dependent tasks are dispatched concurrently and what measures should be taken in the event of a failure.

4 Recovery-Conscious Scheduling

The goal of recovery-conscious scheduling (RCS) is to ensure system availability even during localized recovery. By recovery-consciousness, we mean that the scheduler must assure availability of resources for normal operation even during a localized recovery process. One way to achieve this objective is to intelligently isolate the recovery process by bounding the amount of resources that will be consumed by the recovering tasks.

Figure 4 shows a performance-oriented scheduling algorithm that does not take recovery dependencies into consideration while scheduling tasks. The diagram shows a 4-way SMP system where each processor independently schedules tasks from the same

job queue. This scheduling algorithm aims at maximizing the throughput and minimizing the response time of user requests, which are internally translated by the system into numerous tasks of three types R1, R2, R3. The ovals represent tasks and the same shading scheme is used to denote tasks that are dependent in terms of recoverability. As shown in Figure 4, when all CPU resources are utilized for concurrently executing the tasks that have failure/recovery dependencies, then failure and subsequent recovery can consume all the resources of the system, stalling other tasks that could have proceeded with normal operation. Moreover, continuing to dispatch additional dependent tasks before the localized recovery process can be completed only further aggravates the problem of unavailability.

4.1 Recovery Groups and Resource Pools

In order to deal with the problem illustrated in Figure 4, we infuse “recovery consciousness” into the scheduler. Our recovery-conscious scheduler will enforce some serialization of dependent tasks thereby controlling the extent of a localized recovery operation that may occur at any time. To formally describe recovery conscious scheduling, we first define two important concepts: **recovery groups** and **resource pools**.

Recovery Groups: A recovery group is defined as the unit of a localized recovery operation i.e., the set of tasks that will undergo recovery concurrently. When clean-up procedures are initiated for any task within a recovery group, all other tasks belonging to the same recovery group that are executing concurrently will also initiate appropriate clean-up procedures in order to maintain the system in a consistent state. Note that recovery groups are determined based on explicitly specified dependencies that are further refined by the system based on observations of implicit dependencies. By definition, every task belongs to a single recovery group. Thus tasks in the system can be partitioned into multiple disjoint recovery groups.

Resource Pools: The concept of resource pools is used as a method to partition the overall processing resources into smaller independent resource units, called **resource pools**. Although we restrict resource pools in this paper to processors, the concept can be extended to any pool of identical resources such as replicas of metadata or data. Recovery conscious scheduling maps resource pools to recovery groups, thereby confining a recovery operation to the resources available within the resource pool assigned to it.

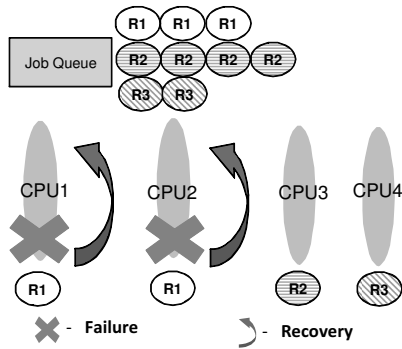


Figure 5: Recovery Oriented Scheduling

4.2 Mapping of Resource Pools to Recovery-Groups

The recovery-conscious scheduling (RCS) algorithms implement the mapping between recovery groups and resource pools. There are different ways that one can map recovery groups to resource pools. The choice of decision depends on the type of trade-offs one would like to make between recovery time and system availability and performance. Static scheduling of resource pools to recovery groups is one end of the spectrum and is only effective in situations where task level dependencies with respect to recoverability are well understood and the workloads of the system is stable. Dynamic scheduling of recovery groups to resource pools represents another end of the spectrum and may better adapt to the changing workload and more effectively utilize resources, but it is more costly in terms of scheduling management. Between the two ends of the spectrum are the partially dynamic scheduling algorithms.

Figure 5 depicts a recovery-conscious scheduler for the same set up as the one used for the performance-oriented scheduler, where tasks are organized into recovery groups – R1 (shaded as fill), R2 (horizontal lines) and R3 (downward diagonal). The processing resources (four CPUs in this example) are organized into three resource pools such that recovery group R1 is mapped to a pool consisting of two processors and recovery groups R2 and R3 are each mapped to a pool consisting of one processor. In case of a failure within group R1, the recovering tasks are now restricted to two of the available four processors so that the other two processors remain available for normal operation. Additionally, the scheduler suspends further dispatching of tasks belonging to group R1 until the localized recovery process completes. This example highlights two aspects of a recovery-conscious scheduler: **proactive** and **reactive**.

Proactive RCS comes into play during normal operation and enhances availability by enforcing some

```

while true do
  repeat
    repeat
      ScanDispatch(HighPriorityQueue)
    until HighPriorityLoopCount
    ScanDispatch(MediumPriorityQueue)
  until MediumPriorityLoopCount
  ScanDispatch(LowPriorityQueue)
end while

```

Figure 6: Qos-based scheduling

```

while true do
  repeat
    repeat
      ScanDispatch(HighPriorityQueue for  $\rho_1$ )
    until HighPriorityLoopCount
    ScanDispatch(MediumPriorityQueue for  $\rho_1$ )
  until MediumPriorityLoopCount
  ScanDispatch(LowPriorityQueue for  $\rho_1$ )
end while

```

Figure 7: Recovery conscious scheduling

degree of serialization of dependent tasks. The goal of proactive scheduling is to reduce the impact of a failure by trying to bound the number of outstanding tasks per recovery group. Then in the event of a failure within any recovery group, the number of tasks belonging to that recovery group that are currently executing and need to undergo recovery are also controlled. By limiting the extent of a recovery process, proactive scheduling can help the system recover sooner, and at the same time, it controls the amount of resources dedicated to the recovery process. Proactive RCS thereby ensures resource availability to normal operation even during a localized recovery process.

The reactive aspect of recovery conscious scheduling takes over *after a failure has occurred*. When localized recovery is in progress, reactive RCS suspends the dispatch of tasks belonging to the group undergoing recovery until the recovery completes. This ensures quick completion of recovery by preventing transitive expansion of the recovery scope and avoiding deadlocks.

4.3 System Considerations

The deployment of recovery conscious scheduling in practice requires the design and implementation of the scheduler to meet the stringent performance requirements of the storage system, sustaining the desired high throughput and low response time. Put differently, recovery-conscious scheduling should offer comparable efficiency in throughput and latency as those provided by performance oriented scheduling.

We outline below some factors that must be taken into consideration while comparing recovery conscious scheduling with performance oriented

scheduling in a multi-core/SMP environment.

Note that our scheduling algorithms are concerned with partitioning resources between tasks belonging to different “components” of the same system which adds a second orthogonal level to the scheduling problem. We continue to respect the QoS or priority considerations specified by the designer at the level of user requests. For example, Figure 6 shows an existing QoS based scheduler using high, medium and low priority queues. Figure 7 shows how recovery-conscious scheduling used by a pool ρ_1 dispatches jobs based on both priority and recovery-consciousness (by picking jobs only from the recovery groups assigned to it).

We use *good-path* and *bad-path* performance as the two main metrics for comparison of the recovery-conscious schedulers with performance oriented schedulers. By ‘good-path’ performance we mean the performance of the system during normal operation. We use the term ‘bad-path’ performance to refer to the performance of the system under localized failure/recovery.

Both good path and bad path performance can be measured using end-to-end performance metrics such as throughput and response time. In addition, we can also measure the performance of a scheduler from system-level factors, including cpu utilization, number of tasks dispatched over time, queue lengths, the overall utilization of other resources such as memory, and the ability to meet service level agreements and QoS specifications.

5 Classification of RCS Algorithms

We classify recovery conscious scheduling (RCS) algorithms based on the method in which resource pools are distributed across recovery groups. As discussed in the previous section, we categorize recovery-conscious scheduling algorithms into three classes: static, partially dynamic, and fully dynamic. This classification represents varying degrees of trade-offs between fault isolation and performance, ranging from static mappings which emphasize recoverability over performance, to different ways of balancing between recoverability and performance, to a completely dynamic mapping of resources to recovery groups, which maximizes the utilization of resources while trying to meet recovery constraints.

In order to provide a better understanding of the design philosophy of our recovery-conscious scheduling, we devise a *running example scenario* that is used to illustrate the design of all three classes of RCS algorithms. This running example has five resource pools: $\rho_1, \rho_2, \rho_3, \rho_4$ and ρ_5 and four recovery groups: $\gamma_1, \gamma_2, \gamma_3$ and γ_4 . We use σ_i to denote the *recoverability constraint* for the recovery group

Recovery Groups	γ_1	γ_2	γ_3	γ_4
% of Workload	40%	20%	20%	20%
Recoverability constraints (σ_i)	2	1	1	1

Table 1: Recovery constraints

Recovery Groups	γ_1	γ_2	γ_3	γ_4
Resource Pools	ρ_1, ρ_2	ρ_3	ρ_4	ρ_5

Table 2: Static mapping

γ_i . Constraint σ_i specifies the upper limit on the amount of resources (processors in this case) that can be dedicated to the recovery group γ_i ($1 \leq i \leq 4$ in our running example). Since we are concerned with processing resources in this paper, it also indicates the number of tasks belonging to a recovery group that can be dispatched concurrently. The recoverability constraint σ_i is determined based on both the recovery group workload i.e., the number of tasks dispatched, and the observed task-level recovery time. Although recoverability constraints are specified from the availability standpoint, they must take performance requirements into consideration in order to be acceptable. Recoverability constraints are primarily used for proactive RCS.

For ease of exposition we assume that all resource pools are of equal size (1 processor each). Table 1 shows the workload distribution between the recovery groups and the recoverability constraint per group, where two processors are assigned to the recovery group γ_1 and one processor is assigned to each of the remaining three groups.

In contrast to the scenario in Table 1 where no resource pools are shared by two or more recovery groups, when more than one recovery group is mapped to a resource pool the scheduler must ensure that the dispatching scheme does not result in starvation. By avoiding starvation, it ensures that the functional interactions between the components are not disrupted. For example in our implementation we used a simple round-robin scheme for each scheduler to choose the next task from different recovery groups sharing the same resource pool. Other schemes such as those based on queue lengths or task arrival time are also appropriate as long as they avoid starvation.

5.1 Static RCS

Static recovery conscious scheduling algorithms construct static mappings between recovery groups and resource pools. The initial mapping is provided to the system based on the observations of the workload and known recoverability constraints, such as previously observed localized recovery times. The

mappings are static in the sense that they do not continuously adapt to changes in resource demands and workload distribution. Table 2 shows a mapping between the pools $\rho_1 \dots \rho_5$ and the recovery groups $\gamma_1 \dots \gamma_4$. This mapping assigns resource pools to recovery groups based on the workload distribution and the recoverability constraints given in Table 1. Concretely, in this mapping recovery group γ_1 is mapped to two pools ρ_1 and ρ_2 . Similarly groups γ_2 , γ_3 and γ_4 are each assigned a single resource pool. Each processor dispatches work only from its assigned recovery group.

This approach aims at achieving strict recovery isolation. As a result, it loses out on utilization of resources, which in turn impacts both throughput and response time. Although this is a naive approach to performing recovery-conscious scheduling it helps us in understanding issues related to the performance and recoverability trade-off. Note that all our RCS algorithms avoid starvation by using a round-robin scheme to cycle between recovery groups sharing the same resource pool. In systems where the workload is well understood and sparse in terms of resource utilization, static mappings offer a simple means of achieving serialization of recovery dependent tasks.

Implementation Considerations: There are two main data structures that are common to all RCS algorithms: (1) the mapping tables and (2) the job queues. Mapping table implementations keep track of the list of recovery groups assigned to each resource pool. They also keep track of groups that are currently undergoing recovery for the purpose of reactive scheduling. In our system we used a simple array-based implementation for mapping tables.

There are a couple of options for implementing job queues. Recall that recovery-consciousness is built on top of the QoS or priority based scheduling. We could use multiple QoS based job queues (for example, high, medium and low priority queues) for each pool or for each group. In our first prototype, we chose the latter option and implemented multiple QoS based job queues for each recovery group for a number of reasons. Firstly, this choice easily fits into the scenario where a single recovery group is assigned to multiple resource pools. Secondly, it offers greater flexibility to modify mappings at run-time. Finally, reactive scheduling (i.e., suspending dispatch of tasks belonging to a group undergoing localized recovery) can be implemented more elegantly as the resource scheduler can simply skip the job queues for the recovering group. Enqueue and dequeue operations on each queue are protected by a lock. An additional advantage of a mapping implemented using multiple independent queues is that it reduces the degree of contention for queue locks.

Recovery Groups	γ_1	γ_2	γ_3	γ_4
Resource Pools	All	All	ρ_4	ρ_5

Table 3: Partial Dynamic RCS: Alternative mapping

```

...
repeat
  workFound := false
  for  $\gamma_i$  in current mapping do
    workFound := ScanDispatch(HighQueue for  $\gamma_i$ )
    if workFound then
      break
    end if
  end for
  if !workFound then
    AcquireLease()
    for  $\gamma_j$  in alternative mapping do
      workFound := ScanDispatch(HighQueue for  $\gamma_j$ )
      if leaseExpired() OR workFound then
        break
      end if
    end for
  end if
until HighPriorityLoopCount
//Similarly for Medium and Low Priority tasks

```

Figure 8: Partial Dynamic RCS

5.2 Partial dynamic RCS

The second class of algorithms are partially dynamic and allow the recovery-conscious scheduler to react (in a constrained fashion though) to sudden spikes or bursty workload of a recovery group.

The main drawback of static RCS is that it results in poor utilization of resources due to the strictly fixed mapping. Partial dynamic RCS attempts to alleviate this problem by using a relatively more flexible mapping of resources to recovery groups, allowing groups to utilize spare resources. Partially dynamic RCS algorithms begin with a static mapping. However, when the utilization is low, the system switches to an alternative mapping that redistributes resources across recovery groups.

For example, with the static mapping of Table 2 with changing distribution of workloads, resources allocated to recovery groups γ_3 and γ_4 may be under utilized while groups γ_1 and γ_2 may be swamped with work. Under these circumstances, the system switches to an alternative mapping shown in Table 3. Now groups γ_1 and γ_2 can utilize spare resources across the system even if this may mean potentially violating their recoverability constraints specified in Table 1. Note that γ_3 and γ_4 still obey their recoverability constraints. In summary, partially dynamic mappings allows the flexibility of selectively violating the recoverability constraints when there are spare resources to be utilized, whereas static mappings strictly obey recoverability constraints.

The aim of the partial dynamic mapping is to improve utilization over static schemes by opening up spare resources to recovery groups with heavy workloads. With the above example although there

is a danger of a single recovery group (for e.g., γ_1) running concurrently across all resource pools, note that this is highly unlikely if other groups have any tasks enqueued for dispatching. There are multiple combinatorial possibilities in designing alternative mappings for partially-dynamic schemes. The choice of which components should continue to stay within their recoverability bounds is to be made by the system designer using prior information about individual component vulnerabilities to failures.

Implementation Considerations: There are two implementation considerations that are specific to the partially dynamic scheduling schemes: (1) the mechanism to switch between initial schedule and an alternative schedule, and (2) the mapping of recovery group tasks to the shared resource pools.

We use a lease expiry methodology to flexibly switch between alternative mappings. Note that the pool schedulers switch to the alternative mapping based on the resource utilization of the current pool. With the partially dynamic scheme, the alternative mappings are acquired under a lease, which upon expiry causes the scheduler to switch back to the original schedule. The lease-timer is set based on observed component workload trends (such as duration of a burst or spike) and the cost of switching. For example, since our implementation had a very low cost of switching between mappings, we set the lease-timer to a single dispatch cycle. Figure 8 shows the pseudo-code for a partial dynamic scheduling scheme using a lease expiry methodology. For the sake of simplicity we do not show the tracking method (round-robin) used to avoid starvation in the scheduler.

Recall from the implementation considerations for the static mapping case that we chose to implement job queues on a per recovery group basis. This allowed for easy switching between the current and alternative mapping which only involves consulting a different mapping table. Task enqueue operations are unaffected by the switching between mappings.

5.3 Dynamic RCS

Dynamic recovery-conscious scheduling algorithms assign recovery groups to resource pools at runtime. In dynamic RCS, tasks are still organized into recovery groups with recoverability bounds specified for each group. However, all resource pools are mapped to all recovery groups. The schedulers cycle through all groups giving preference to groups that are still within their recoverability bounds, i.e., occupying fewer resources than specified by the bound. If no such group is found, then tasks are dispatched while trying to minimize the resource consumption by any individual recovery group.

This class of algorithms aim at maximizing utilization of resources at the cost of selectively violating the recoverability constraints. Note that all recovery-conscious algorithms are still designed to perform reactive scheduling, i.e., suspend the dispatching of tasks whose group is currently undergoing localized recovery. The aspect that differentiates the various mapping schemes is the proactive handling of tasks to improve system availability. The dynamic scheme can be thought of as trying to use load balancing among recovery groups in order to achieve both recovery isolation and good resource utilization.

Implementation Considerations: A key implementation consideration specific to dynamic RCS is the problem of keeping track of the number of outstanding tasks belonging to each recovery group. We maintain this information in a per-processor data structure that keeps track of the current job.

Recall that implementing job queues on the per recovery-group basis helps us implement dynamic mappings efficiently and flexibly. One of the critical optimizations for dynamic RCS algorithms involves understanding and mitigating the scheduling overhead imposed by the dynamic dequeuing process. In on going work we are conducting experiments with different setups to characterize this overhead. However our results in this paper show that even with the additional scheduling cost dynamic RCS schemes perform very well both under good-path and bad-path conditions.

6 Experiments

We have implemented our recovery-conscious scheduling algorithms on a real industry-standard storage system. Our implementation involved no changes to the functional architecture. Our results show that dynamic RCS can match performance oriented scheduling under good path conditions while significantly improving performance under failure recovery.

6.1 Experimental Setup

Our algorithms were implemented on a high-capacity, high-performance and highly reliable enterprise storage system built on proprietary server technology due to which some of the setup and architecture details presented in this paper have been desensitized. The system is a storage facility that consists of a storage unit with two redundant 8-way server processor complexes (controllers), memory for I/O caching, persistent memory (Non-Volatile Storage) for write caching, multiple FCP, FICON or ESCON adapters connected by a redundant high bandwidth (2 GB) interconnect, fiber channel disk drives, and

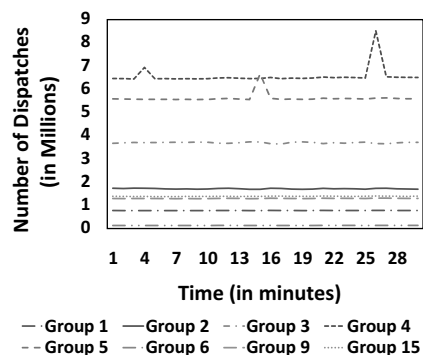


Figure 9: Cache-Standard

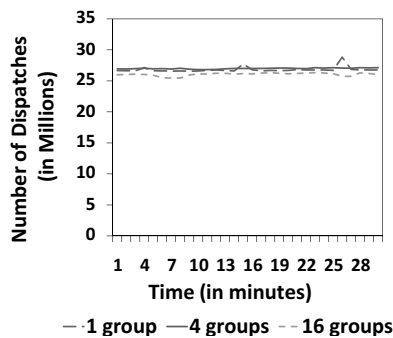


Figure 10: Efficiency vs Recovery groups

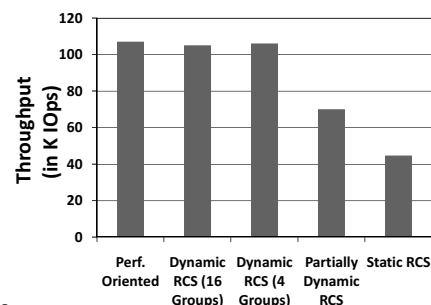


Figure 11: Good path throughput

management consoles. The system is designed to optimize both response time and throughput. The basic strategy employed to support continuous availability is the use of redundancy and highly reliable components.

The embedded storage controller software is similar to the model presented in Section 2. The software is also highly-reliable with provisions for quick recovery (under ~ 6 seconds) at the system-level. The system has a number of interacting components which dispatch a large number of short running tasks. For the experiments in this paper based on programmer explicit recovery dependency specifications, we identified 16 recovery groups which roughly correspond to functional components such as cache manager, device manager, SCSI command processor etc. However, some recovery groups may perform no work in certain workloads possibly due to features being turned off. We chose a pool size of 1 CPU which resulted in 8 pools of equal size. The system already implements high, medium and low priority job queues. Our recovery-conscious scheduling implementation therefore uses three priority based queues per recovery group. For the partially dynamic case, based on the workload we have identified two candidates for strict isolation - groups 4 and 5. For the static mapping case each recovery group is mapped to resource pools proportional to its ratio of the total task workload.

6.2 Workload

We use the z/OS Cache-Standard workload [1, 14] to evaluate our algorithms. The z/OS Cache-standard workload is considered comparable to typical online transaction processing in a z/OS environment. The workload has a read/write ratio of 3, read hit ratio of 0.735, destage rate of 11.6% and a 4K average transfer size. The setup for the cache-standard workload was CPU-bound. Figure 9 shows the number of tasks dispatched per-recovery group under the work-

load over 30 minutes. Group 4 has the highest task workload (~ 6.5 M tasks/min) followed by group 5 (~ 5 M/min). Eight of the groups which have nearly negligible workload are not visible in the graph. We use this workload to measure throughput and response times. While measuring cpu utilization we only count time actually spent in task execution and do not include time spent acquiring queue locks, dequeuing jobs or polling for work.

6.3 Experimental Results

We compare RCS and performance oriented scheduling algorithms using good-path (i.e. normal condition) and bad-path (under failure recovery) performance.

6.3.1 Effect of additional job queues

We first performed some benchmarking experiments to understand the effect of additional job queues on the efficiency of the scheduler. Using the cache-standard workload, we measured the aggregate number of dispatches per minute with varying number of recovery groups - 16, 4 and 1 (which is identical to performance-oriented scheduling) to measure scheduler efficiency with dynamic RCS. The four and one recovery group cases were implemented by collapsing multiple groups into a single larger group. Recall that each recovery group results in three job queues for high, medium and low priority jobs. Figure 10 shows the aggregate number of tasks dispatched per minute with 1, 4 and 16 recovery groups. As the figure shows the number of dispatches are almost identical in the three cases ($\pm 2\%$). Although more job queues imply having to cycle through more queue locks while dispatching work, increasing the number of job queues reduces contention for queue locks both when enqueueing and dequeuing tasks. For most of the experiments in this paper we choose a configuration with 16 recovery groups.

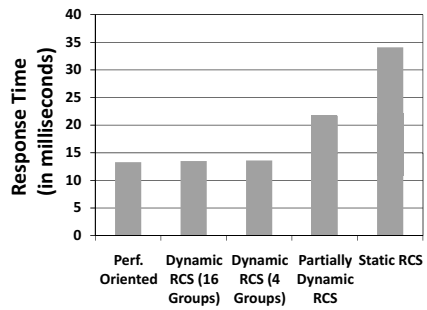


Figure 12: Good path latency

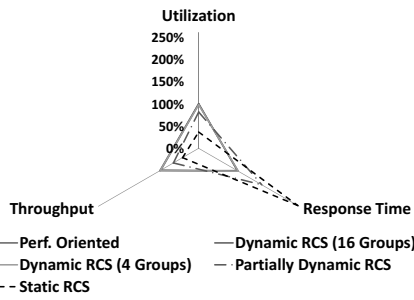


Figure 13: CPU utilization

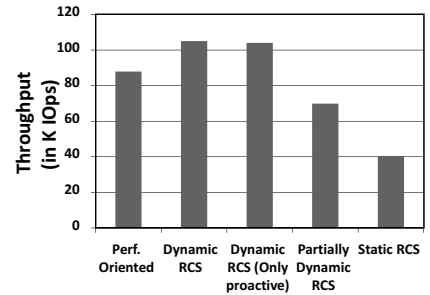


Figure 14: Bad path throughput

6.3.2 Good-path Performance

Recovery-conscious scheduling can be an acceptable solution only if it is able to meet the stringent performance requirements of a high-end system. In this experiment we compare the good-path (i.e. under normal operation) performance of our RCS algorithms with the existing performance-oriented scheduler.

Figure 11 shows the good-path throughput for the performance-oriented and recovery-conscious scheduling algorithms. The average throughput for the dynamic RCS case with 16 groups (105 KIOps) and 4 groups (106 KIOps) was close to that for the performance-oriented scheduler (107 KIOps). On the other hand, with partially dynamic RCS, the system throughput drops by nearly 34% (~ 69.9 KIOps), and with static RCS by nearly 58% (~ 44.6 KIOps) compared to performance oriented scheduling.

Figure 12 compares the response time with different RCS schemes and performance-oriented scheduling. Again, the average response-time for the dynamic RCS case with 16 recovery groups (13.5 ms) and 4 recovery groups (13.6 ms) is close to the performance-oriented case (13.3 ms). However, with the partially-dynamic RCS scheduling, the response time increases by nearly 63% (21.7 ms) and by 156% with static RCS (34.1ms).

Both the throughput and response time numbers can be explained using the next chart, Figure 13. The radar plot shows the relationship between throughput, response time and cpu-utilization for each of the cases. As the figure shows, the cpu utilization has dropped by about 19% with partially dynamic RCS and by 63% with static RCS. The reduction in cpu utilization eventually translates to reduced throughput and increased response time in a cpu-bound workload intensive environment. These numbers seem to indicate that in such an environment schemes that reduce the utilization can result in significant degradation of the overall performance. However note that the normal operating

range of many customers may be only around 6-7000 IOps [22]. If that be the case, then partially-dynamic schemes can more than meet the system requirement even while ensuring some recovery isolation.

6.3.3 Bad-path Performance

Next, we compare RCS algorithms with performance-oriented scheduling under bad-path or failure conditions. In order to understand the impact on system throughput and response time when localized recovery is underway, we inject faults into the cache standard workload. We choose a candidate task belonging to recovery group 5 and introduce faults at a fixed rate (1 for each 10000 dispatches). Recovery was emulated and recovery from each fault was set to take approximately 20 ms. On an average this introduces an overhead of 5% to aggregate execution time per minute of the task. During localized recovery, all tasks belonging to the same recovery group that are currently executing in the system and that are dispatched during the recovery process also experience a recovery time of 20 ms each. We measured performance (throughput and latency) averaged over a 30 minute run.

In the case of recovery-conscious scheduling algorithms, reactive scheduling kicks in when any group is undergoing recovery. Under those circumstances, tasks belonging to that recovery group already under execution are allowed to finish, but further dispatch from that group is suspended until recovery completes.

Figure 14 shows the average system throughput with fault injection. The average throughput using only performance oriented scheduling (87.8 KIOps) drops by nearly 16.3% when compared to dynamic RCS (105 KIOps) that also uses reactive policies. On the other hand dynamic RCS continues to deliver the same throughput as under normal conditions. Note that this is still not the worst case for performance oriented scheduling. In the worst case, all resources may be held up by the recovering tasks

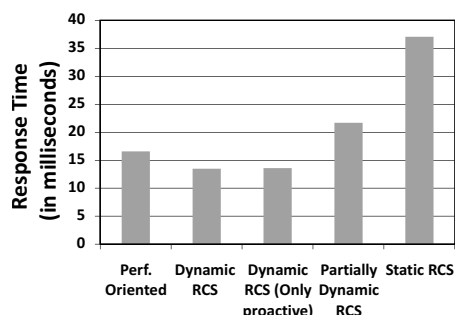


Figure 15: Bad path latency

resulting in actual service outage and the problem would only worsen with increasing localized recovery time and system size.

The figure also compares proactive and reactive policies in dynamic RCS. The results show that with only proactive scheduling we are able to sustain a throughput (104 KIOps) which is just $\sim 1\%$ less than that using both proactive and reactive policies (105 KIOps).

The graph also compares partially dynamic RCS (69.9 KIOps) and static RCS (40.4 KIOps). While these schemes are able to sustain almost the same throughput as they do under good path, overall, the performance of these schemes results in 20% and 54% drop in throughput respectively compared to performance oriented scheduling.

Figure 15 compares the latency under bad-path code with different scheduling schemes. Compared to dynamic RCS (13.5 ms), performance oriented scheduling (16.6 ms) results in a 22.9% increase in response time. At the same time, even without reactive scheduling, dynamic RCS (13.6 ms with only proactive) increases response time by only 0.7%. Again, partially dynamic RCS (21.7ms) and static RCS (37.1 ms) result in latency close to their good path performance but which is still too high when compared to dynamic RCS.

We performed experiments with other configurations of dynamic, partially dynamic and static schemes and using other workloads too. However due to space constraints we only present key findings from those experiments. In particular we used a disk-bound internal workload (and hence low cpu utilization of about $\sim 25\%$) to study the effect of our scheduling algorithms under a sparse workload. We used the number of task dispatches as a metric of scheduler efficiency. The fault injection mechanism was similar to the cache-standard workload, however due to the workload being sparse, we introduced an overhead of only 0.3% to the aggregate execution time of the faulty recovery group. Our results showed that dynamic RCS was able to achieve

as many dispatches as performance oriented scheduling under good path operation and increase the number of dispatches by 0.7% under bad-path execution. With partial dynamic RCS dispatches dropped by 20% during good path operation and by only 3.9% during bad path operation compared to performance oriented scheduling. The same static mapping used in the cache standard workload when run in this new environment resulted in the system not coming up. While this may be due to setup issues, it is also likely that insufficient resources were available to the platform tasks during start-up. We are investigating further on a more appropriate static mapping for this environment.

6.4 Discussion

The fact that recovery-conscious scheduling requires minimal change to the software allows for it to be easily incorporated even in legacy systems.

Dynamic RCS can match good path performance of performance oriented scheduling and at the same time significantly improve performance under localized recovery. Even for the small 5% recovery overhead introduced by us, we could witness a 16.3% improvement in throughput and a 22.9% improvement in response time with dynamic RCS. Moreover, the qualitative benefits of RCS in enhancing availability and ensuring that localized recovery is scalable with system size makes it an interesting possibility as systems are moving towards more parallel architectures. Our experiments with various scheduling schemes (some not reported in this paper) have given us some insights into the overhead costs such as lock spin times imposed by RCS algorithms. In ongoing work we are continuing to characterize and investigate further optimizations to RCS schemes.

Our results also seem to indicate that for small localized recovery time and system sizes, proactive policies i.e. mapping resource pools to recovery groups, can deliver the advantage of recovery-consciousness. However as system size increases or localized recovery time increases, we believe that the actual benefits of reactive policies such as suspending dispatch from groups undergoing recovery may become more pronounced. In ongoing research we are experimenting with larger setups and longer localized recovery times.

Static and partial dynamic RCS schemes are limited by their poor resource utilization in workload intensive environments. Hence we do not recommend these schemes in an environment where the system is expected to run at maximum throughput. However, the tighter qualitative control that these schemes offer may make them, especially partially dynamic RCS, more desirable in less intensive environments where there is a possibility to over-

provision resources, or when the workload is very well understood. Besides in environments where it ‘pays’ to isolate some components of the system from the rest such mappings may be useful. We are continuing research on optimizing these algorithms and understanding properties that would prescribe the use of such static or partially dynamic schemes.

7 Related Work

Our work is largely inspired by previous work in the area of software fault tolerance and storage system availability. Techniques for software fault tolerance can be classified into fault treatment and error processing. Fault treatment aims at avoiding the activation of faults through environmental diversity, for example by rebooting the entire system [6, 24], micro-rebooting sub-components of the system [2], through periodic rejuvenation [13, 5] of the software, or by retrying the operation in a different environment [17]. Error processing techniques are primarily checkpointing and recovery techniques [7], application-specific techniques like exception handling [21] and recovery blocks [19] or more recent techniques like failure-oblivious computing [20].

In general our recovery conscious approaches are complementary to the above techniques. However we are faced with several unique challenges in the context of embedded storage software. First, the software being legacy code rules out re-architecting the system. Second, the tight coupling between components makes both micro-reboots and periodic rejuvenation tricky. Rx [17] demonstrates an interesting approach to recovery by retrying operations in a modified environment but it requires checkpointing of the system state in order to allow ‘rollbacks’. However given the high volume of requests (tasks) experienced by the embedded storage controller and their complex operational semantics, such a solution may not be feasible in this setup.

Failure-oblivious computing [20] introduces a novel method to handle failures - by ignoring them and returning possibly arbitrary values. This technique may be applicable to systems like search engines where a few missing results may go unnoticed, but is not an option in storage controllers.

The idea of localized recovery has been exercised by many. Transactional recovery using checkpointing/logging methods is a classic topic in DBMSs [16] and is a successful implementation of fine-grained recovery. In fact application-specific recovery mechanisms such as recovery blocks [19], and exception handling [21] are used in almost every software system. However, very few have made an effort on understanding the implications of localized recovery on system availability and performance in a multi-core environment where interacting tasks are exe-

cuting concurrently. Likewise, the idea of recovery-conscious scheduling is to raise the awareness about localized recovery in the resource scheduling algorithms to ensure that the benefits of localized recovery actually percolate to the level of system availability and performance visible to the user. Although vast amounts of prior work have been dedicated to resource scheduling, to the best of our knowledge, such work has mainly focused on performance [25, 11, 12, 8, 4]. Also much work in the virtualization context has been focused on improving system reliability [18] by isolating VMs from failures at other VMs. In contrast, our development focuses more on improving system availability by distributing resources *within* an embedded storage software system by identifying fine-grained recovery scopes. Compared to earlier work on improving storage system availability at the RAID level [23], we are concerned with the embedded storage software reliability. These techniques are at different levels of the storage system and are complementary.

8 Conclusion and Future Work

In this paper we presented a recovery conscious framework for multi-core architectures and techniques for improving the resiliency and recovery efficiency of highly concurrent embedded storage software. Our main contributions include a task-level recovery model and the development of recovery-conscious scheduling, a non-intrusive technique to reduce the ripple effect of software failure and improve the availability of the system. We presented a suite of RCS algorithms and quantitatively evaluated them against performance oriented scheduling. Our evaluation showed that dynamic RCS can significantly improve performance under failure recovery while matching performance oriented scheduling during normal operation.

In order to adopt RCS for large software systems, a significant challenge is to identify efficient recovery scopes. In ongoing work we are working on developing more generic guidelines that would assist in identifying fine-grained recovery scopes. Even with pluggable mechanisms like RCS it is necessary to emphasize that high-availability should still be a design concern and not an after-thought. We hope our framework would encourage developers to incorporate additional error handling and anticipate more error scenarios and that our scheduling schemes would aid in scaling efficient error handling with system size.

9 Acknowledgments

The authors would like to express their gratitude to David Whitworth, Andrew Lin, Juan Ruiz (JJ),

Brian Hatfield, Chiahong Chen and Joseph Hyde for helping us perform experimental evaluations and interpret the data. We would also like to thank K.K.Rao, David Chambliss, Brian Henderson and many others in the Storage Systems group at IBM Almaden Research Center who provided us with the resources to perform our experiments and provided valuable feedback. We thank our anonymous reviewers, our shepherd Dr. Mary Baker and Prof. Karsten Schwan for the useful comments and feedback that have helped us improve the paper.

This work is partially supported by an IBM PhD scholarship and an IBM Storage Systems internship for the first author, and the NSF CISE IIS and CSR grants, an IBM SUR grant, as well as an IBM faculty award for the authors from Georgia Tech.

References

- [1] Ibm z/architecture principles of operation. *SA22-7832*, IBM Corporation, 2001.
- [2] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: a soft-state system case study. *Perform. Eval.*, 56(1-4):213–248, 2004.
- [3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot—a technique for cheap recovery. *OSDI*, 2004.
- [4] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA*, pages 105–115, New York, NY, USA, 2007. ACM Press.
- [5] S. Garg, A. Puliafito, M. Telek, and K. Trivedi. On the analysis of software rejuvenation policies. In *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97)*, 1997.
- [6] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.
- [7] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, October 1992.
- [8] A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *SIGMETRICS*, pages 13–24, New York, NY, USA, 2007. ACM Press.
- [9] M. Hartung. IBM totalstorage enterprise storage server: A designer's view. *IBM Syst. J.*, 42(2):383–396, 2003.
- [10] HP. HSG80 array controller software.
- [11] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.
- [12] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, 2005.
- [13] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS*, page 381, Washington, DC, USA, 1995. IEEE Computer Society.
- [14] L. LaFrese. Ibm totalstorage enterprise storage server model 800 new features in lic level 2.3.0 (performance white paper). *ESS Performance Evaluation*, IBM Corporation, 2003.
- [15] I. Lee and R. K. Iyer. Software dependability in the tandem guardian system. *IEEE Trans. Softw. Eng.*, 21(5):455–467, 1995.
- [16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.
- [17] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies — a safe method to survive software failure. In *SOSP*, Oct 2005.
- [18] H. Ramasamy and M. Schunter. Architecting dependable systems using virtualization. In *Workshop on Architecting Dependable Systems in conjunction with 2007 International Conference on Dependable Systems and Networks (DSN-2007)*, 2007.
- [19] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM Press.
- [20] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebe. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, Berkeley, CA, USA, 2004. USENIX Association.
- [21] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *SP*, pages 273–280, Washington, DC, USA, 2007. IEEE Computer Society.
- [22] Sirius. Sirius enterprise systems group disk drive storage hardware. *Solicitation EPS050059-A4*.
- [23] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. *ACM Transactions on Storage (TOS)*, 1(2):133–170, May 2005.
- [24] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *FTCS*, pages 2–9, 1991.
- [25] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, 2006.

Improving I/O Performance of Applications through Compiler-Directed Code Restructuring

Mahmut Kandemir Seung Woo Son
Department of Computer Science and Engineering
The Pennsylvania State University
{kandemir,sson}@cse.psu.edu

Mustafa Karakoy
Department of Computing
Imperial College
m.karakoy@imperial.edu.uk

Abstract

Ever-increasing complexity of large-scale applications and continuous increases in sizes of the data they process make the problem of maximizing performance of such applications a very challenging task. In particular, many challenging applications from the domains of astrophysics, medicine, biology, computational chemistry, and materials science are extremely data intensive. Such applications typically use a disk system to store and later retrieve their large data sets, and consequently, their disk performance is a critical concern. Unfortunately, while disk density has significantly improved over the last couple of decades, disk access latencies have not. As a result, I/O is increasingly becoming a bottleneck for data-intensive applications, and has to be addressed at the software level if we want to extract the maximum performance from modern computer architectures.

This paper presents a compiler-directed code restructuring scheme for improving the I/O performance of data-intensive scientific applications. The proposed approach improves I/O performance by reducing the number of disk accesses through a new concept called disk reuse maximization. In this context, disk reuse refers to reusing the data in a given set of disks as much as possible before moving to other disks. Our compiler-based approach restructures application code, with the help of a polyhedral tool, such that disk reuse is maximized to the extent allowed by intrinsic data dependencies in the application code. The proposed optimization can be applied to each loop nest individually or to the entire application code. The experiments show that the average I/O improvements brought by the loop nest based version of our approach are 9.0% and 2.7%, over the original application codes and the codes optimized using conventional schemes, respectively. Further, the average improvements obtained when our approach is applied to the entire application code are 15.0% and 13.5%, over the original application codes and the codes optimized using

conventional schemes, respectively. This paper also discusses how careful file layout selection helps to improve our performance gains, and how our proposed approach can be extended to work with parallel applications.

1 Introduction

In the recent past, large scale applications in science and engineering have grown dramatically in complexity. As a result, scientists and engineers expend great effort to implement software systems that carry out these applications and interface them with the instruments and sensors that generate data. Apart from their huge computational needs, these large applications have tremendous I/O requirements as well. In fact, many scientific simulations tend to generate huge amounts of data that must be stored, mined, analyzed, and evaluated. For example, in a combustion application [39], features based on flame characteristics must be detected and tracked over time. Based upon evolution, simulations need to be steered in different regions, and different types of data need to be stored for further analysis. A simulation involving three-dimensional turbulent flames involving detailed chemistry can easily result in 5 tera-bytes of data being stored on disk, and the total storage requirement can be in the order of peta-bytes when one considers the fact that numerous such simulations have to be performed to reach meaningful and accurate conclusions. Other scientific applications have also similar storage and I/O requirements.

Unfortunately, as far as software – in particular compilers – are concerned, I/O has always been neglected and received much less attention in the past compared to other contributors to an application's execution time, like CPU computation, memory accesses and inter-CPU communication. This presents an important problem, not just because modern large-scale applications have huge I/O needs, but also the progresses in storage hardware are not in the scale that can meet these pressing I/O demands.

Advances in disk technology have enabled the migration of disk units to 3.5-inch and smaller diameters. In addition, the storage density of disks has grown at an impressive 60 percent annually, historically, and has accelerated to greater than a 100 percent rate since 1999 [20]. Unfortunately, disk performance has not kept pace with the growth in disk capacities. As a result, I/O accesses are among primary bottlenecks in many large applications that store and manipulate large data sets. Overall, huge increases in data set sizes combined with slow improvements in disk access latencies motivate for software-level solutions to the I/O problem. Clearly, this I/O problem is most pressing in the context of data-intensive scientific applications, where increasingly larger data sets are processed.

While there are several ways of improving I/O behavior of a large application, one of the promising approaches has been cutting the number of times the disks are accessed during execution. This can be achieved at different layers of the I/O subsystem and be attacked by using caching which keeps frequently used data in memory (instead of disks) or by restructuring the application code in a way that maximizes data reuse. While both the approaches have been explored in the past [2, 3, 9, 10, 16, 21, 24], the severity of the I/O problem discussed above demands further research. In this paper, we focus on a compiler-directed code restructuring for improving I/O performance of large-scale scientific applications that process disk-resident data sets. A unique advantage of the compiler is that it can analyze an entire application code, understand global (application wide) data and disk access patterns (if data-to-disk mapping is made available to it), and – based on this understanding – restructure the application code and/or data layout to achieve the desired performance goal. This is a distinct advantage over pure operating system (OS) based approaches that employ rigid, application agnostic optimization policies as well as over pure hardware based techniques that do not have the global (application wide) data access pattern information. However, our compiler based approach can also be used along with OS and hardware based schemes, and in fact, we believe that this is necessary to reach a holistic solution to the growing I/O problem.

The work presented in this paper is different from prior studies that explore compiler support for I/O in at least two aspects. First, our approach can optimize the entire program code rather than individual, parallel loop-nests, as has been the case with the prior efforts. That is, as against to most of the prior work on compiler-directed I/O optimization, which restructure loops independent of each other, our approach can restructure the entire application code by capturing the interactions among different loop nests. An advantage of this is that our approach

does not perform a local (e.g., loop nest based) optimization which is effective for the targeted scope but harmful globally. However, if desired, our approach can be applied to individual loop nests or functions/subprograms independently. Second, we also discuss the importance of file layout optimization and of adapting to parallel execution. These two extensions are important as 1) the results with our layout optimization indicate that additional performance savings (7.0% on average) are possible over the case code re-structuring is used alone, and 2) the results with the multi-CPU extension show that this extension brings 33.3% improvement on average over the single-CPU version.

The proposed approach improves I/O performance by reducing the number of disk accesses through *disk reuse maximization*. In this context, disk reuse refers to reusing the data in a given set of disks as much as possible before moving to other disks. Our approach restructures the application code, with the help of a polyhedral tool [26], such that disk reuse is maximized to the extent allowed by intrinsic data dependencies in the code. We can summarize the major contributions of this paper as follows:

- We present a compiler based disk reuse optimization technique targeting data intensive scientific applications. The proposed approach can be applied at the loop nest level or whole application level.
- We discuss how the success of our approach can be increased by modifying the storage layout of data, and how it can be extended to work under parallel execution.
- We present an experimental evaluation of the proposed approach using seven large scientific applications. The results collected so far indicate that our approach is very successful in maximizing disk reuse, and this in turn results in large savings in I/O latencies. More specifically, the average I/O improvements brought by the loop nest based version of our approach are 9.0% and 2.7%, over the original application codes and the codes optimized using conventional schemes, respectively. Further, the average improvements obtained when our approach is applied to the entire application code are 15.0% and 13.5%, over the original application codes and the codes optimized using conventional schemes, respectively.

The rest of this paper is organized as follows. The next section explains the disk system architecture assumed by our compiler. It also presents the key concepts used in the remainder of the paper. Section 3 gives the mathematical details behind the proposed compiler-based approach. Section 4 discusses how our approach can be extended by taking accounts of the storage layout of data. Section 5 gives an extension to capture the disk access interactions among the threads of a parallel application. An experimental evaluation of our approach and a comparison with the conventional data reuse optimization scheme are pre-

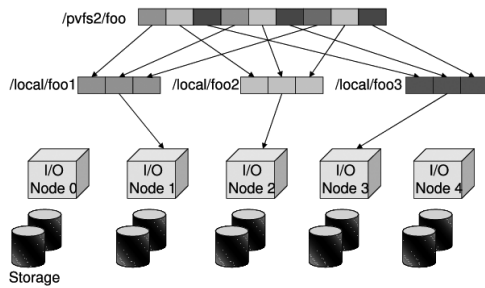


Figure 1: Striping a file over parallel disks. Striping is performed at two levels, the first of which can be exposed to and controlled by software.

sented in Section 6. Section 7 discusses related work and Section 8 concludes the paper by summarizing its main contributions and discussing briefly possible future extensions.

2 Disk System Architecture and Importance of Disk Reuse

Figure 1 depicts the disk system architecture targeted by our work. In many high-performance storage systems today, there are two levels of *striping*. The first one, which is at the software level, divides an array into equal-sized blocks (stripes) and distributes these blocks across a number of I/O nodes in a round-robin fashion. The second level of striping occurs at an I/O node level where the data blocks mapped to an I/O node are further striped (at a much finer granularity) over the disks managed by that I/O node (e.g., using one of the RAID schemes [4]). While this second level of striping is not visible to the software, the first level of striping is; and in fact, many modern file systems provide hints that can be used to query or control some of the striping parameters (e.g., the number of I/O nodes to be used for striping data, the I/O node from which the striping begins, and the size of a stripe). The compiler-based disk reuse optimization approach presented in this work focuses on this software-level striping. In our discussion, we assume a single disk per I/O node, and therefore, we use the terms “I/O node” and “disk” interchangeably as long as the context is clear.

We also assume that a portion of the main memory of the computation node is reserved to serve as *buffer* (also called *cache*) for frequently used disk data. If a requested data item is found in this cache, no disk I/O is performed and this can reduce data access latencies significantly. While it is possible to employ several buffer management schemes, the one used in this work operates under the LRU policy which replaces the least recently used stripe when a new block is to be brought in. Selection of the buffer management scheme to employ is orthogonal to the main focus of this paper. It is impor-

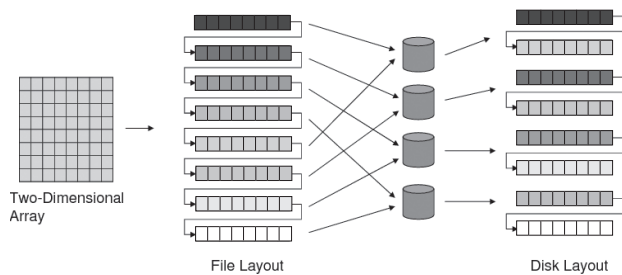


Figure 2: Different mappings a two-dimensional disk-resident array goes through. In the most general case, the memory layout, file layout and disk layout for an array can be all different from each other. We use D to represent a mapping from file layout to disk layout.

tant to note that all the disks in the system share the same buffer (in the computation node) to cache their data, and thus, effective management of this buffer is very critical. Note also that, in addition to the cache in the computation node, the I/O nodes themselves can also employ caches. Our optimization target in this paper, however, is the performance of the cache in the computation node.

Figure 2 shows the *mappings* a two-dimensional data array data goes through as far as disk system storage is concerned. Array data in memory is stored in file using some storage order, which may be row-major, column-major, or in a blocked fashion. This is called the *file layout*. (Note that this may be different from the memory layout adopted by the underlying programming language. For example, a C array can be stored in file using column-major layout as opposed to row-major, which is the default memory layout for multi-dimensional arrays in C.) The file is then striped across the available disks on the system. Therefore, two data elements which are neighbors in the memory space can get mapped to separate disks as a result of this series of mappings. Similarly, data blocks that are far apart from each other can get mapped to the same disk as a result of striping. In this work, when we use the term “disk-resident array,” we mean an array that is mapped to the storage system using these mappings. Note that, while it is also possible to map multiple data arrays to one file or one data array to multiple files, in this work we consider only one-to-one mappings between data arrays and files. However, our approach can easily be extended, if desired, to work with one-to-many or many-to-one mappings as well. Unless otherwise stated, all the data arrays mentioned in this paper are disk resident.

Let us now discuss why disk reuse is important and how an optimizing compiler can improve it. The next section gives the technical details of our proposed compiler-based approach to improving disk reuse. Recall that disk reuse means using a data in a given set of

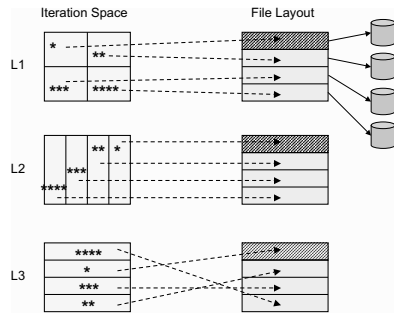


Figure 3: An example access patterns and the corresponding file layout. We restructure the application code such that, once a particular file block (stripe) is accessed, iterations that access the same block extracted from all loop nests (L1, L2, and L3) are executed together. Note that all three loop nests access the same disk-resident array.

disks as much as possible before accessing other disks.

- When disk reuse is improved, the chances of finding the requested data in the buffer increases. As a simple scenario, consider a case in which a given disk resident array is divided into four stripes and each stripe is stored on a separate disk. We can expect a very good buffer performance if the code can be restructured such that accesses to a given disk are clustered together. This is because such a clustering improves chances for catching data in the buffer at the time of reuse. Figure 3 illustrates this scenario. In this scenario, three different loop nests (L1, L2, and L3) access a given disk-resident array. Figure 3 shows which portions of the iteration spaces of these loop nests access what stripes (we assume 4 stripes). In a default execution, the iteration space can be traversed in a row-wise fashion. As a result, for example, when the first row of L1 is executed, two stripes are accessed (and they compete for the same buffer in the computation node). In our approach however the iteration spaces are visited in a buffer-aware fashion. If dependencies allow, we first execute the chunks (marked using *) from L1, L2, and L3 (one after another). Note that all these chunks (iterations) use the same stripe (and therefore achieve a very good data reuse in the buffer). After these, the chunks marked ** are executed, and these are followed by the chunks marked ***, and so on.

- Since our approach clusters disk accesses to a small set of disks at any given time (and maximizes the number of unused disks), in a storage system that accommodates power-saving features, unused disks can be placed into a low-power operating mode [25, 30, 40]. However, in this paper we do not quantify the power benefits of our approach.

As mentioned earlier, our work focuses on I/O inten-

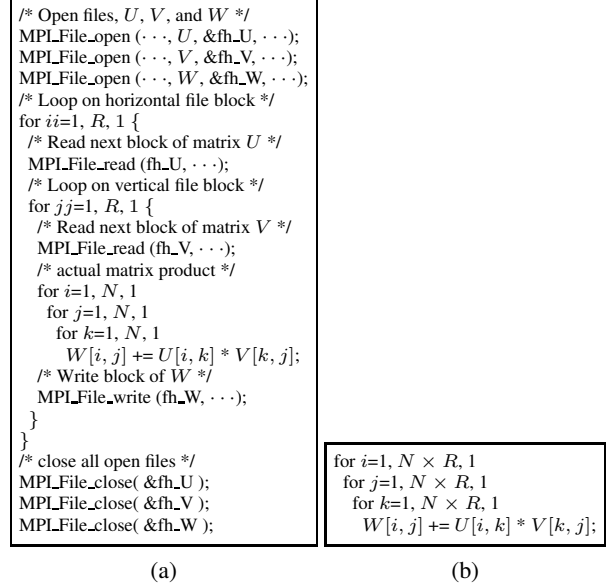


Figure 4: (a) A matrix multiplication code written in MPI-IO that operates on disk resident arrays. In this example code, each file is divided into $R \times R$ blocks and each block has $N \times N$ elements. (b) The corresponding simplified version that omits the file I/O commands and highlights computation.

sive applications that process disk resident arrays. The application codes we target are written in MPI-IO [33], which is the part of the MPI library [12] that handles file I/O related activities. MPI-IO allows synchronous and asynchronous file reads and writes as well as a large set of collective file operations. Figure 4(a) shows an MPI-IO code fragment that performs matrix multiplication on disk resident arrays. For clarity reasons, in our discussion, we omit the MPI-IO commands and represent such a code as shown in Figure 4(b). That is, all the code fragments discussed in this paper are assumed to have the corresponding file I/O commands.

3 Mathematical Details

To capture disk accesses and optimize them, we use polyhedral algebra based on Presburger Arithmetic. Presburger formulas are made of arithmetic and logic connectives and existential (\exists) and universal (\forall) quantifiers. In our context, we used them to capture and enumerate loop iterations that exhibit disk access locality. We use the term *disk map* to capture a particular set of disks (I/O nodes) in the system. For a storage system with T disks, we use $\Lambda = \lambda_1 \lambda_2 \lambda_3 \dots \lambda_T$ to indicate a disk map. As an example, if $T = 4$, $\lambda_1 \lambda_2 \lambda_3 \lambda_4 = 0110$ represents a subset (of disks) that includes only the second and third disks, whereas 1110 specifies a subset that includes all disks

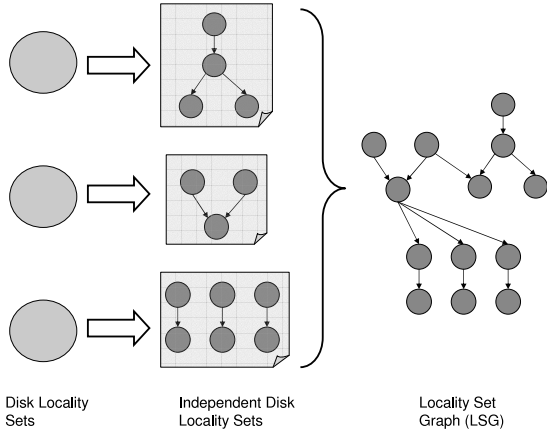


Figure 5: An example that shows three disk locality sets, their independent locality sets, and an LSG. Each independent LSG in the middle corresponds each disk locality set on the left. Edges in the independent LSGs represent the dependencies between the nodes. The LSG on the right is generated by combining each independent LSG taking accounts of dependencies.

except the last one. Assuming that array-to-disk mapping (e.g., such as one shown in Figure 2) is exposed to the compiler, the compiler can set up a relationship between the loop iterations in an application and the disks in the storage system the corresponding (accessed) array elements are stored. For example, if array reference $U[i+1]$ appears in a loop with iterator i , for a given value of i we can determine the disk that stores the corresponding array element ($U[i+1]$).

We define *disk locality set* as a set of loop iterations – which may belong to any loop nest in the application code – that access the set of disks represented by the same disk map. Mathematically, for a given disk map Λ , we can define the corresponding disk locality set (\mathcal{Q}_Λ) as:

$$\mathcal{Q}_\Lambda = \{\vec{\xi} \mid \vec{\xi} \in \mathcal{I} \wedge \{\exists R \in \mathcal{R} \text{ such that } D(R(\vec{\xi})) \in \Lambda\}\},$$

where \mathcal{I} represents the set of all loop iterations (coming from all nests) in the program; $\vec{\xi}$ is a particular loop iteration; \mathcal{R} represents the set of all references to disk-resident arrays; $R(\cdot)$ is a reference (a mapping from the loop iterations to the data elements), and $D(\cdot)$ is a disk mapping (striping) function, which maps the data elements to the disks in the system. We use expression $D(R(\vec{\xi})) \in \Lambda$ to indicate that the data element accessed via $R(\vec{\xi})$ is mapped to one of disks in the set represented by disk map Λ .

Let us give an example to illustrate the disk locality set concept. Assume that a disk-resident array U of size K is striped over 4 disks with a stripe of size $K/4$ (i.e., each

disk has a single stripe). Assume further that, for the sake of illustration, we have a single loop i that iterates from 1 to $K-2$ and uses two references, $U[i]$ and $U[i+2]$, to access this disk-resident array. In this case, we have:

$$\begin{aligned} \mathcal{Q}_{1100} &= \{\xi \mid [1 \leq \xi \leq K-2] \\ &\quad \wedge \{[1 \leq \xi \leq K/2] \vee [1 \leq \xi+2 \leq K/2]\}\} \\ &= \{\xi \mid 1 \leq \xi \leq K/2-2\}, \end{aligned}$$

which gives us the set of iterations that access only the first two disks. (Note that, the $\{[1 \leq \xi \leq K/2] \vee [1 \leq \xi+2 \leq K/2]\}$ part is due to two references to array U , and since the loop nest has only a single loop, we use ξ instead of $\vec{\xi}$.)

An important characteristic of the iterations that belong to the same \mathcal{Q}_Λ is that they exhibit a certain degree of locality as far as disks are concerned. As a result, if, somehow, we can transform the application code and execute iterations that belong to the same \mathcal{Q}_Λ successively, we can improve disk reuse (as in the case illustrated in Figure 3). However, this is not very trivial in practice because of two reasons. First, the inter-iteration data dependencies in the application code may not allow such an ordering, i.e., we may not be able to restructure the code for disk reuse and (at the same time) maintain its original semantics. Second, even if such an ordering is legal from the viewpoint of data dependencies, it is not clear how it can be obtained, i.e., what type of code restructuring can be applied to obtain the desired ordering. More specifically, it is not clear whether the transformation (code restructuring) requested for clustering accesses to a subset of disks can be obtained using a combination of well-known transformations such as loop fusion, loop permutation, and iteration space tiling [36]. From a compiler angle, there is nothing much to do for the first reason. But, for the second one, polyhedral algebra can be of help, which is investigated in the rest of the paper.

Suppose, for now, that the application code we have has no dependencies (we will drop this assumption shortly). In this case, we may be able to improve disk reuse (and the performance of the buffer in the computation node) using the following two-step procedure:

- For any given Λ , execute iterations in the \mathcal{Q}_Λ set consecutively, and
- In moving from \mathcal{Q}_Λ to $\mathcal{Q}_{\Lambda'}$, select Λ' such that the Hamming Distance between Λ and Λ' is minimum when all possible Λ' s are considered.

The first item above helps us have good disk reuse by executing the iterations that belong to the subset of disks represented by a given disk map. The second item, on the other hand, helps us minimize the number of disks whose status (i.e., being used or not being used) has to be changed as we move from executing the iterations in \mathcal{Q}_Λ

to executing the iterations in $Q_{\Lambda'}$. As a result, by applying these two rules repeatedly, one can traverse the entire iteration space in a disk-reuse efficient manner, and this in turn helps improve the performance of the buffer.

However, real I/O-intensive applications typically have lots of data dependencies and, thus, the simple approach explained above will not suffice in practice. We now discuss how the compiler can capture the dependencies that occur across the different disk locality sets.

We start by observing that the iterations in a given disk locality set Q_{Λ} can have data dependencies amongst themselves. We consider a partitioning (such a partitioning can be obtained using the Omega library [26] or similar polyhedral tools) of Q_{Λ} into subsets $Q_{\Lambda,1}, Q_{\Lambda,2}, \dots, Q_{\Lambda,s}$ such that $Q_{\Lambda,i} \cap Q_{\Lambda,j} = \emptyset$ for any i and j , $Q_{\Lambda,1} \cup Q_{\Lambda,2} \cup \dots \cup Q_{\Lambda,s} = Q_{\Lambda}$, and for any i and j , all data dependencies are either from $Q_{\Lambda,i}$ to $Q_{\Lambda,j}$ or from $Q_{\Lambda,j}$ to $Q_{\Lambda,i}$. The first two of these constraints indicate that the subsets are disjoint and collectively cover all the iterations in Q_{Λ} , and the last constraint specifies that, as far as Q_{Λ} is concerned, the iterations in any $Q_{\Lambda,i}$ can be executed successively without any need of executing an iteration from the set $Q_{\Lambda} - Q_{\Lambda,i}$. That is, when we start executing the first iteration in $Q_{\Lambda,i}$, all the remaining iterations in $Q_{\Lambda,i}$ can be executed one after another (of course, these iterations can have dependencies among themselves). We refer to any such subset $Q_{\Lambda,i}$ of Q_{Λ} as the “independent disk locality set,” or the “independent set” for short. As an example, Figure 5 shows three locality sets (on the left) and the corresponding independent locality sets (in the middle). The first locality set in this example contains four independent locality sets, and these independent locality sets are connected to each other using three dependence edges. In our approach, independent locality sets are the building blocks for the main – graph based – data structure used by the compiler for disk reuse optimization.

This graph, called the “locality set graph” or LSG for short, can be defined as $LSG=(V, E)$ where each element of V represents an independent disk locality set, and the edges in E capture the dependencies between the elements of V . In other words, an LSG has the $Q_{\Lambda,i}$ sets as its nodes and the dependencies among them as its edges. The right portion of Figure 5 shows an example LSG. The question then is to schedule the nodes of the LSG while preserving the data dependency constraints between the nodes. What we mean by “scheduling” in this context is determining an order at which the nodes of the graph will be visited (during execution). Clearly, we want to determine such a schedule at compile time and execute it at runtime, and the goal of this scheduling should be minimizing the Hamming Distance as we move from one independent set to another. For the example LSG in Figure 5, we show in Figure 6 two legal

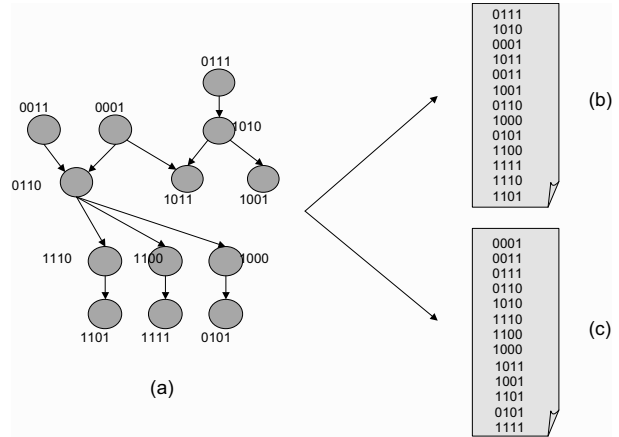


Figure 6: An example LSG (a) and two legal schedules (b and c). The order of the schedule (c) is determined based on minimum Hamming Distance, thereby exhibiting less number of disk state changes.

(dependence preserving) schedules. Note that the total (across all time steps) Hamming Distance for the first schedule (in Figure 6(b)) is 28, whereas that for the second one (in Figure 6(c)) is 15. Therefore, we can expect the second one to result in a better disk buffer reuse than the first one.

However, we note that a given LSG may *not* always be schedulable as it is. This is because it can have cycles involving a subset of its nodes. Consider for example the example LSG shown in Figure 7(a). This LSG has two cycles, and it is not possible to determine a schedule for it. In order to convert a non-schedulable LSG to a schedulable one, we somehow have to break all the cycles it contains. But, before explaining how this can be done, we want to discuss briefly the reasons for the cycles in an LSG. There are two reasons for cycles in a given LSG. First, for a given Q_{Λ} , there can be a cycle formed by its independent sets (the $Q_{\Lambda,i}$ s) only. Second, the independent sets coming from the different disk locality sets can collectively form a cycle, i.e., two independent disk locality sets such as $Q_{\Lambda,i}$ and $Q_{\Lambda',j}$ can involve in the same cycle, where $\Lambda \neq \Lambda'$.

If an LSG has one or more cycles, we need to find a way of eliminating those cycles before the LSG can be scheduled for improving disk reuse. In the rest of this section, we discuss our solution to this issue. It can be observed that there are at least two ways of removing a cycle from a cyclic LSG. First, the nodes that are involved in the cycle can be combined into a single node. This technique is called *node merging* in this paper, and is illustrated in Figures 7(b) and (c), for the cyclic LSG in Figure 7(a). Note that, when the nodes are merged, the iterations in the combined node can be executed in an or-

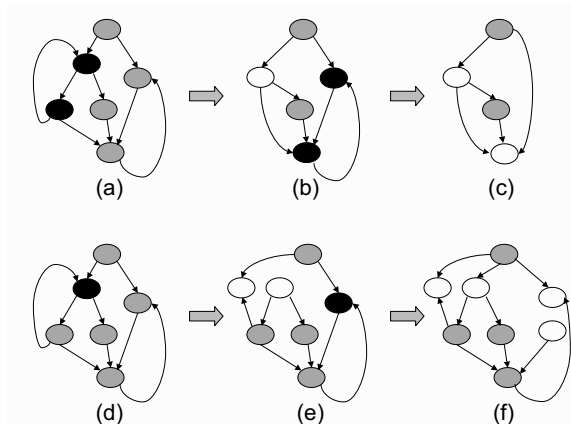


Figure 7: Application of node merging (a through c) and that of node splitting (d through f). In each case, the nodes selected for applying node merging and splitting are shown in black color, while the affected nodes are shown in white color.

der that respect data dependencies. The second technique that can be used for breaking cycles is *node splitting* (see Figures 7(d) through (f)). While these techniques can be used to convert a non-schedulable LSG to a schedulable one, each has a potential problem we need to be aware of. (clearly, one can also use a combination of node merging and node splitting to remove cycles.) A consequence of node merging is that the corresponding iteration execution may not be very good, as the two successively executed iterations (from the combined node) can access different set of disks. (note that the disk map of the merged node is the bitwise-OR of the disk maps of the involved nodes.) That is, the potential cost of eliminating cycles is a degradation in disk reuse. Node splitting on the other hand has a different problem. It needs to be noted that not all splittings can help us eliminate the cycles in a given LSG. In other words, one needs to be careful in deciding which iterations to place into each of the resulting sub-nodes so that the cyclic dependence at hand can be broken. Determination of these iterations may not be very trivial in practice, but is doable using automated compiler analysis supported by a polyhedral tool. Also, after splitting, the disk maps of the resulting nodes can be determined based on the loop iterations they contain. It is to be observed that node splitting in general also increases the code size as we typically need a separate nest for each node in the LSG.

Our preliminary experience with these two techniques showed that in general node splitting is preferable over node merging, mainly because the latter can lead to significant losses in disk reuse, depending on the application code being optimized. Therefore, in our analysis below, we restrict our discussion to node splitting only. However, as mentioned above, code size can be an is-

sue with node splitting, and hence, we keep the number of splittings at minimum. So, the problem now becomes one of *determining the minimum number of nodes to split that makes the LSG cycle free*. We start by noting that splitting a node, if done successfully, has the same effect as that of *removing* a node (and the arrows incident on it) from the graph. That is, as far as removing cyclic dependencies is concerned, node splitting and node removal are interchangeable. Fortunately, this latter problem (removing the minimum number of nodes from a graph to make it cycle free) has been studied in the past extensively and is known as the “feedback vertex set” problem [8]. Karp was the first one to show that this problem is NP-complete on directed graphs; but it is known today that the undirected version is also NP-complete. Moreover, the problem remains NP-complete for directed graphs with no in-degree or out-degree exceeding 2, and also for planar directed graphs with no in-degree or out-degree exceeding 3. Fortunately, there exist several heuristic algorithms proposed in the literature for the feedback vertex set problem. In this work, we use the heuristic discussed in [7]. Since the details of this heuristic are beyond the scope of this paper, we do not discuss them.

As an example, Figure 8(a) gives a sample LSG. Figure 8(b) highlights the node selected by the heuristic in [7], and Figure 8(c) gives the pruned LSG. Note that splitting the node identified by [7] eliminates both the cycles. Figure 8(d) on the other hand shows the LSG after node splitting, which is cycle free. It is important to note that, while node removal and splitting obviously result in different LSGs, their effects on the schedulability of a given graph are similar; that is, both of them make a given cyclic graph schedulable. In particular, the set of nodes returned by the heuristic in [7] is the set of *minimum* nodes that need to be considered for splitting (though, as explained below, in some cases we may consider more nodes for potential split). Based on this discussion, Figure 10 gives the algorithm used by our compiler for restructuring a given code for improving disk reuse. This algorithm starts by building the initial LSG for the input code. This LSG can contain cycles, and hence, we next invoke procedure `remove_cycles(.)` to obtain a cycle free LSG. While this step uses the heuristic approach in [7], it needs to do some other things as well, as explained below.

In the rest of this section, we discuss details of our node splitting strategy. Once a node is identified (using the heuristic in [7]) as a potential candidate for splitting, our approach checks whether it can be split satisfactorily. What is meant by “satisfactorily” in this context is that, although in theory we can always a split a node into two or more nodes, the one we are looking for has the properties explained below.

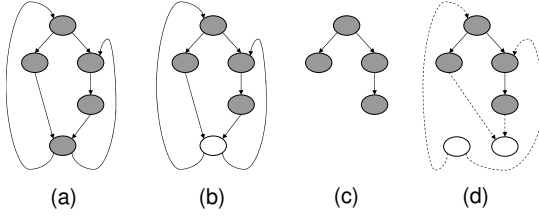


Figure 8: (a) An example LSG. (b) The node selected by the algorithm in [7] for eliminating cycles. (c) The graph after the cycles have been eliminated. (d) The graph after the node detected by the algorithm in [7] is split into two nodes.

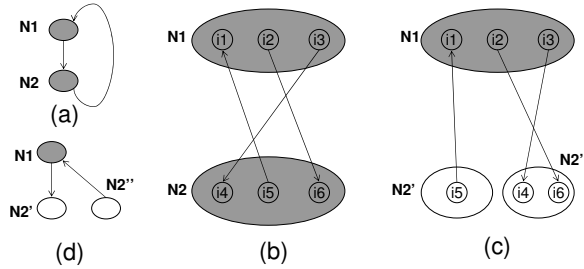


Figure 9: (a) An example LSG. (b) Details of dependencies of the LSG. The node N1 contains the iterations i1 through i3 whereas the node N2 contains the iterations i4 through i6. (c) Detailed view after splitting. (d) The graph after splitting. In (b), iterations i4 and i6 are the iterations that have incoming dependence from the node N1, the iteration i5, on the other hand, is the iteration that has outgoing dependence edge to the node N1.

Assume that Q_Λ is the node to be split. Let \mathcal{G}_I be the set of nodes from which there are dependencies to node Q_Λ . That is, for each member, $Q_{\Lambda'}$, of \mathcal{G}_I , there is at least a dependence from $Q_{\Lambda'}$ to Q_Λ . Assume further that \mathcal{G}_O is the set of nodes to which we have dependencies from node Q_Λ . In other words, we have a dependence from Q_Λ to each node, $Q_{\Lambda''}$, of \mathcal{G}_O . Suppose now that Q_Λ is split into two sub-nodes: Q_{Λ_a} and Q_{Λ_b} . We call this split “satisfactory” if all of the following three conditions are satisfied after the split:

- No dependency goes from any $Q_{\Lambda'} \in \mathcal{G}_I$ to Q_{Λ_b} . In other words, all the original in-coming dependencies of Q_Λ are directed to Q_{Λ_a} .
- No dependency goes from Q_{Λ_b} to any $Q_{\Lambda''} \in \mathcal{G}_O$. In other words, all the original out-going dependencies of Q_Λ are directed from Q_{Λ_a} .
- No dependency exists between Q_{Λ_a} and Q_{Λ_b} .

We say that cycle in question is removed (if the removal of Q_Λ is sufficient to remove the cycle; otherwise, the other nodes in the cycle, which is detected by the algorithm in [7], have to be visited), if the three conditions above are satisfied. It needs to be emphasized that, pre-

cisely speaking, the last condition above is not always necessary. However, if there exist dependencies between Q_{Λ_a} and Q_{Λ_b} , it is possible that we still have cycle(s) in the LSG due to Q_Λ , depending on the direction of these dependencies. Enforcing the last condition, along with the others, ensures that the cyclic dependencies are removed completely. Figure 9 illustrates an example LSG. Figure 9(a) shows an original LSG in coarse grain and (b) illustrates the dependencies between the two nodes of this LSG in fine grain. Assuming that the bottom node has been selected for removal (ultimately one of possible nodes to be split) by the heuristic in [7], Figures 9(c) and (d) show the result of splitting LSG in fine grain and coarse grain, respectively. As another example, Figure 8(d) shows the split version of the LSG in Figure 8(a). Note that, due to the third condition above, our approach may need to look at more nodes than ones determined by the heuristic described in [7]. We also need to mention that, in our implementation, these three conditions listed above are checked using the Omega library [26].

4 File Layout Modification

So far in our discussion we considered only data access pattern restructuring for improving disk reuse. It is to be noted however that data layout on disks can also play an important role as far as disk reuse is concerned. Specifically, a different file layout can lead to a different disk layout which can in turn lead to a different amount of disk reuse. Let us consider the following code fragment:

```
for  $i=1, N, 1$ 
  for  $j=1, N, 1$ 
     $U[i, j] = f(V[j, i]);$ 
```

In this code fragment, two disk-resident arrays are accessed (as mentioned earlier, we do not show explicit I/O statements). While one of these is accessed in row wise, the second one is traversed column wise. Consequently, storing both the arrays in the same fashion in file (e.g., as shown in Figure 2) may not be the best option since such a storage will not be able to take advantage of disk reuse for the second array as its data access pattern and file storage pattern would not match. Now, consider the file layout transformation depicted in Figure 11. If this transformation is applied to the second array (V) in the code fragment above, we can expect better disk reuse.

The important question to address is to select the mapping that maximizes disk reuse. We start by noting that the search space is very large for potential file layout transformations, as there are many ways of transforming a file layout. However, our experience with disk-intensive scientific applications and our preliminary experiments suggest that we can restrict the potential map-

```

MAX — maximum number of node splitting operations allowed;
QΛ — disk locality sets;
dep(i, j) — returns TRUE if dependence between QΛ,i and QΛ,j exists;
H(QΛ,i, QΛ,j) — returns Hamming distance between QΛ,i and QΛ,j;
VS — set of QΛ,is that are ready to schedule;
QΛ,x — last scheduled locality set;

procedure build_LSG() {
  for each QΛ,i {
    build independent disk locality sets;
  }
  for any two subsets QΛ,i and QΛ,j ∈ QΛ {
    if (dep(i, j) == TRUE) add edge between two nodes, QΛ,i and QΛ,j;
  }
}

procedure remove_cycles() {
  split_count = 0;
  while (split_count < MAX) {
    apply node splitting;
    split_count++;
  }
}

procedure schedule_LSG() {
  VS = ∅;
  for each QΛ,i ∈ LSG {
    if (QΛ,i has no parents || ∀ parents of QΛ,i has been scheduled) {
      VS = VS ∪ QΛ,i;
    }
  }
  while (VS ≠ ∅) {
    select QΛ,y ∈ VS such that H(QΛ,x, QΛ,y) is minimum;
    schedule the selected QΛ,y;
    LSG = LSG - QΛ,y; /* remove QΛ,y from LSG; */
    VS = VS - QΛ,y; /* update VS */
    set QΛ,x to QΛ,y; /* update last scheduled locality set */
  }
}

main() {
  call build_LSG();
  if (exists cycles in LSG)
    call remove_cycles();
  while (LSG ≠ ∅) {
    call schedule_LSG();
  }
}

```

Figure 10: Compiler algorithm for scheduling a given code to increase disk reuse. Our algorithm starts by building the LSG and then removes the cycles in the graph if there are any. After obtaining cycle-free LSG, we schedule each node in the graph such that the next node scheduled has the minimum Hamming Distance from the current node. Note that, if desired, this algorithm can be applied to smaller code segments (e.g., a loop nest) as well, instead of the whole program.

pings to dimension permutations. What we mean by “dimension permutation” in this context is reindexing the dimensions of the disk resident array. As an example, restricting ourselves to dimension re-indexings, a three-dimensional disk-resident array can have 6 different file layouts. Let us use $D' = DP$ to represent the disk mapping function when file layout modification is considered, where D is the original disk mapping function discussed earlier in Section 3 and P is a permutation matrix (that implements dimension re-indexing). For exam-

ple, the file layout mapping shown in Figure 11 can be expressed using the transformation matrix

$$P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Note that, for an m -dimensional disk-resident array, P is $m \times m$. (for example, elements on (i, j) data space is mapped to (j, i) as a result of applying the P matrix shown above to a two-dimensional array.)

It is to be noted, however, that the decision for selection of a permutation should be made carefully by considering all the statements that access the array in question. This is because different statements in the application code can access the same disk-resident array using entirely different access patterns, and a layout transformation that does not consider all of them may end up with one that is not good when considered globally. Our file layout selection algorithm is a *profile based* one. In this approach, the application code is profiled by instrumenting it and attaching a set of counters to each disk resident array. For an m -dimensional array, we have $m! + 1$ counters, each keeping the number of times a particular file layout is preferred (note that the total number of possible dimension permutations is $m!$ and one additional counter is used for representing other file layout preferences such as diagonal layouts, for which we do not perform any optimization). In this work, we implement only dimension permutation because other file layouts such as diagonal layouts or blocked layouts are hardly uniform across all execution. Therefore, we do not take any actions for such layouts. At the end of profiling, the layout preference with the largest counter value is selected as the file layout for that array. Figure 12 gives the pseudo code for our file layout selection algorithm. As an example, let us assume that there are three loop nests (with the same number of iterations) accessing the same data array stored in a file. Assume further that the profiling reveals that the first and third loop nests exhibit column-major access pattern whereas the second loop nest exhibits row-major access pattern. As the column-major file layout is preferred more (that is, it will have a higher counter value), we select the corresponding permutation matrix and convert the file layout accordingly.

5 Parallel Execution

It is also important to study disk reuse under parallel execution. An important challenge in this case is to coordinate the disk accesses coming from multiple threads. We note that, even if the disk accesses from individual threads exhibit disk reuse, this does not necessarily mean that the overall execution will have disk reuse. The example in Figure 13 shows a scenario with two threads.

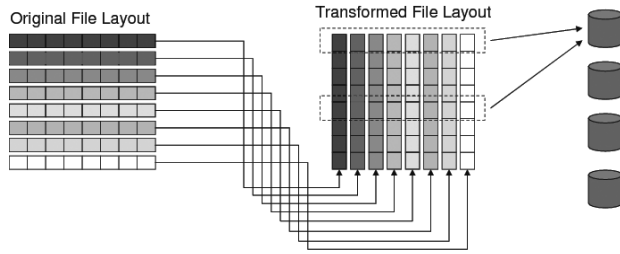


Figure 11: Converting file layout of a two-dimensional disk resident array. In this example, the original file layout in row-major order is transformed into the layout in column-major order.

```

N — number of arrays;
Ui — accessed arrays, where  $1 \leq i \leq N$ ;
Di — original disk mapping for array Ui;
Dim[i] — dimension of array Ui;
Pn — group of permutation matrices;

for i = 1 to N {
    Dim[i] = dimension of array Ui;
}

C[i][Dim[i]+1]; /* counters for each array, Ui, for profiling */

/* perform profiling */
for each array Ui {
    for each statement accessing Ui {
        detect the file layout of accessing Ui;
        select Pj for the determined file layout; /* Pj is jth entry of Pn */
        C[i][j]+; /* increase corresponding counters */
    }
}

for each array Ui {
    select Pj that has the highest C[i][j] value;
    apply D'i = Di Pj; /* transform file layouts */
}

```

Figure 12: Compiler algorithm for transforming file layouts to improve disk reuse. Our algorithm is based on profiling that reveals the most desirable access patterns for each array across all statements within a program.

Assuming the LSGs shown in Figure 13(a), a possible scheduling is given in Figure 13(b). The overall disk reuse (when considering both the thread) in this case is not very good, though scheduling for each thread exhibits high reuse when considered alone. We now consider the alternate scheduling illustrated in Figure 13(c). In this scheduling, the overall disk reuse is very good, which is achieved by scheduling the node in individual thread such that, when both threads execute the selected nodes (at the same scheduling step), the number of disks used is minimized. Note that, within a thread, a node that is scheduled next is chosen based on the minimum Hamming Distance. By adapting this schedule, we do not have any scheduling step in Figure 13(c) that uses all four disks at the same time, whereas, in Figure 13(b), steps 1 and 4 have full usage of all disks, which is not good as far as the buffer (cache) utilization is concerned.

Our scheduling algorithm for an architecture with P

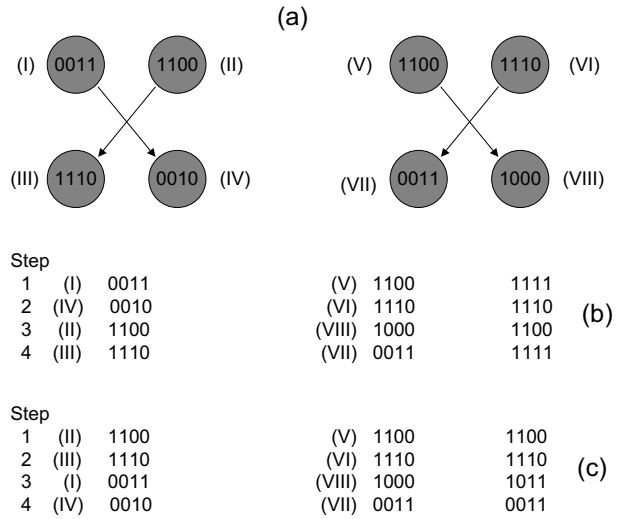


Figure 13: (a) LSGs for two threads of the same application. (b-c) Two legal schedules. The last columns in (b-c) are the disk status two threads, which are obtained by bitwise-ORing the disk status of two threads.

processors and D disks is given in Figure 14. This algorithm takes the LSGs as input, and determines, for each thread, the schedule of the nodes considering the global (inter-thread) usage of the disks. It uses a D -bit global variable G to represent the current usage of the disks. It schedules a node that is ready to be scheduled for each thread that finishes its current task. At each step, the algorithm first tries to schedule the node whose disk requirement can be satisfied with the set of disks currently being used. If multiple nodes satisfy this criterion, we select the one that requires the maximum number of disks to make full utilization of the currently used disks. If such a node does not exist, our algorithm schedules the node whose tag is the closest (in terms of Hamming Distance) to G , the bit pattern that represents the current disk usage (i.e., the disk usage at that particular point in scheduling). This is to minimize the number of disks whose (active/idle) states need to be changed. We want to mention that, since each node may have different execution latency it is possible that the targeted disk reuse (across threads) may not always be achieved. However, we can expect the resulting disk reuse (and buffer performance) to be better than what a random (but legal) scheduling would achieve.

Before moving to the discussion of our experimental results, we want to point out the tradeoff between disk reuse and performance. The parallel version of our approach tries to maximize the disk reuse, and this in turn tends to attract the accesses to a small set of disks. Consequently, in theory, this can lead to performance problems, as the effective disk parallelism is reduced. While

```

P — number of processors;
D — number of disks available;
G — global disk usage map with D bits;
QΛ — disk locality sets;
H(QΛ,i) — returns Hamming distance between QΛ,i and G;
VS[i] — set of QΛ,is that are ready to schedule in processor i;
LSG[i] — LSG for each processor i;

procedure schedule_LSG_P() {
  for i = 1 to P { /* find schedulable QΛ,j from each processor i */
    VS[i] = ∅; /* init each VS[i] of processor i */
    for each QΛ,j ∈ LSG(i) {
      if (QΛ,j has no parents || ∀ parents of QΛ,j has been scheduled) {
        VS[i] = VS[i] ∪ QΛ,j;
      }
    }
  }
  /* schedule all QΛ,j ∈ VS[k] */
  while (VS[k] != ∅) {
    for each processor k {
      select QΛ,i ∈ VS[k] such that it requires the maximum number of
        disks to fully utilize the currently used disks, or H(QΛ,i) is minimum;
      schedule the selected QΛ,i;
      LSG(k) = LSG(k) - QΛ,i; /* remove QΛ,i from LSG(k); */
      VS[k] = VS[k] - QΛ,i;
      update G by bitwise-ORing with Λ of QΛ,i;
    }
  }
}

main() {
  /* build LSG for a code assigned to each processor */
  for i = 1 to P {
    call build_LSG(i);
    if (LSG has cycle(s)) {
      call remove_cycles();
    }
  }

  initialize G by setting all bits to 0;
  while (exists (∃i, LSG[i] != ∅)) {
    call schedule_LSG_P();
  }
}

```

Figure 14: Compiler algorithm for scheduling parallel execution of a given code to increase disk reuse. We use the same procedures given in Figure 10 for building LSG and removing cycles.

this negative impact has already been accounted for in our experiments (discussed in the next section), we observed that its magnitude is not very high. However, this magnitude is typically a function of the application access pattern and disk system parameters as well, and further studies are needed to reach better evaluations.

6 Experiments

6.1 Setup

We implemented our compiler algorithm using the SUIF infrastructure [13]. Our disk reuse optimization increased compilation times of the original applications by about 55% on average (the largest compilation time when our optimization is applied was 87 seconds). When the file layout optimization is enabled, the largest compilation time jumped to 116 seconds. Extension for parallelism added another 9 seconds on average.

Table 1: System parameters.

Parameter	Value
CPU	
Model	Intel P4
Clock Frequency	2.6 GHz
Memory System	
Model	Rambus DRAM
Buffer Capacity	1 GB
Disk System	
Number of I/O Nodes	8
Data Striping	Uses all 8 I/O nodes
Stripe Size	64 KB
Interface	ATA
Storage Capacity/Disk	40 GB
RPM	10,000
Interconnect	
Model	Ethernet
Bandwidth	100 Mbps

We performed our experiments using a platform which includes MPI-IO [33] on top of the PVFS parallel file system [27]. PVFS is a parallel file system that stripes file data across multiple disks in different nodes in a cluster. It accommodates multiple user interfaces which include MPI-IO, traditional Linux interface, and the native PVFS library interface. In all our experiments, we used the MPI-IO interface. Table 1 gives the values of our major experiment parameters. We want to emphasize however that later we present results from our *sensitivity analysis* where we change the default values of some of the important parameters.

For each benchmark in our experimental suite, we performed experiments with different versions:

- **Base Scheme:** This represents the original code without any data locality optimization.
- **Conventional Locality Optimization (CLO):** This represents a conventional data locality optimization technique that employs loop restructuring. It is not designed for I/O, and does not take disk layout into account. The specific data reuse optimizations used include loop interchange, loop fusion, iteration tiling and unrolling. This version in a sense represents the state-of-the-art as far as data locality optimization is concerned.
- **Disk Reuse Optimization – Loop Based (DRO-L):** This is our approach applied at a loop nest level; i.e., each loop nest is optimized in isolation.
- **Disk Reuse Optimization – Whole Program Based (DRO-WP):** This is our approach applied at a whole program level.

All the versions use the MPI-IO interface [33] of PVFS [27] for performing disk I/O. Note that both DRO-L and DRO-WP are the different versions of our approach, and CLO represents the state-of-the-art as far as optimizing data locality is concerned. The reason that we make experiments with the DRO-L and DRO-WP versions separately is to see how much additional benefits one can obtain by going beyond the loop nest level

Table 2: Our applications.

Application Name	Brief Description	Data Set Size (GB)	Disk I/O Time (sec)	Total Time (sec)
sar	Synthetic Aperture Radar Kernel	21.1	64.6	101.4
hf	Hartree-Fock Method	53.6	98.3	173.6
apsi	Pollutant Distribution Modeling	49.9	101.2	238.7
wupwise	Physics/Quantum Chromo-dynamics	27.9	270.3	404.7
e.elem	Finite Element Electromagnetic Modeling	66.2	99.2	191.9
astro	Analysis of Astronomical Data	58.3	171.6	276.4
contour	Contour Displaying	58.7	198.6	338.4

in optimizing for I/O. In addition to these versions, we also implemented and conducted experiments with the file layout optimization scheme discussed in Section 4 and with the parallel version of our approach explained in Section 5.

The unit of buffer (cache) management in our implementation is a data block, and its size is the same as that of a stripe. The set of applications used in this study is given in Table 2. These applications are collected from different sources and their common characteristic is that they are disk-intensive. *apsi* and *wupwise* are similar to their Spec2000 counterparts [14], except that they operate on disk-resident data. The second column briefly explains each benchmark, and the third column gives the total (disk resident) data set size processed by each application. The fourth and fifth columns give the disk I/O times and total execution times, respectively, under the *base scheme* explained above. Note that both the base version and the CLO version are already optimized for buffer usage. In addition, the CLO version is optimized for data locality using conventional techniques, as explained above. Therefore, the performance improvements brought by our approaches (DRO-L and DRO-WP) over these schemes (base and CLO) are due to the code re-ordering we apply. We also see from Table 2 that the contribution of disk I/O times to overall execution times varies between 42.4% and 63.7%, averaging on 57.4%. Therefore, reducing disk I/O times can be very useful in practice. In the remainder of this section, we present and discuss the performance improvements brought by our approach. The disks I/O time savings and overall execution time savings presented below are with respect to the fourth and fifth columns of Table 2, respectively.

6.2 Results

We start by presenting the percentage improvements in disk I/O times brought by the three optimized versions explained above. The results shown in Figure 15 indicate that the average improvements brought by CLO, DRO-L and DRO-WP over the base scheme are 10.4%, 16.1% and 23.9%, respectively. Overall, we see that, while DRO-L performs better than CLO, by 6.3% on average, the best savings are obtained – for all benchmarks tested – with the DRO-WP version, on average, 15.0% and

9.3% over CLO and DRO-L, respectively, meaning that going beyond a single loop nest is important in maximizing the buffer performance. While these improvements in disk I/O times are important, we also need to look at the savings in overall execution times, which include the computation times as well. These results are presented in Figure 16 and show that the average improvements with the CLO, DRO-L and DRO-WP versions are 5.9%, 9.0% and 13.5%, respectively. To better explain how our approach achieves much more performance improvements over conventional data reuse optimization, we present in Figure 17 the average number of times a given data block is visited under the different schemes (that is, how many times a given data block (on average) is brought from disks to cache). We observe that this number is much lower with the DRO-WP version, explaining the additional performance benefits it brings. In fact, on average, the number of disk traversals per block is 3.9 and 2.1 with the base version and DRO-WP, respectively.

Sensitivity Analysis. In this section, we study the sensitivity of our performance savings to several parameters. A critical parameter of interest is the buffer (cache) size. Recall that the default buffer size used in our experimental evaluation so far was 1GB. The graph in Figure 18 gives the results using different buffer sizes. Each point in this graph represents the average value (performance improvement in overall execution time), for a given version, when all seven benchmarks are considered. As expected, the performance gains brought by our approach get reduced when increasing the buffer size. However, even with the largest buffer size we used, the average improvement we have (with the DRO-WP version) is about 6.7%, underlining the importance of disk reuse optimization for better performance. Considering the fact that data set sizes of disk-intensive applications keep continuously increasing, one can expect the disk reuse based approach to be more effective in the future. To elaborate on this issue further, we also performed experiments with larger data sets. Recall that the third column of Table 2 gives the data set sizes used in our experiments so far. Figure 19 gives the average performance improvements, for 1GB and 4GB buffer sizes and two sets of inputs. SMALL refers to the default dataset sizes given in Table 2, and LARGE refers to larger datasets, which are 38.2GB, 66.3GB, 82.1GB, 38.0GB, 88.1GB, 73.7GB, and 81.8GB for *sar*, *hf*, *apsi*, *wupwise*, *e.elem*, *astro* and *contour*, respectively. We see that our approach performs better with larger data set sizes. This is because a larger data set puts more pressure on the buffer, which makes effective utilization of buffer even more critical.

The next parameter we study is the stripe size, which can also be changed using a PVFS call when creating the file. The performance improvement results with different stripe sizes are presented in Figure 20. Our observation is

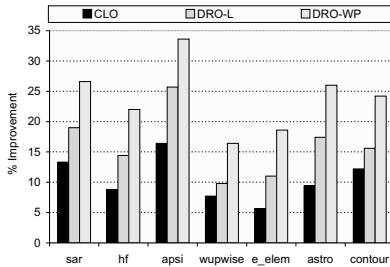


Figure 15: Performance improvements in I/O times.

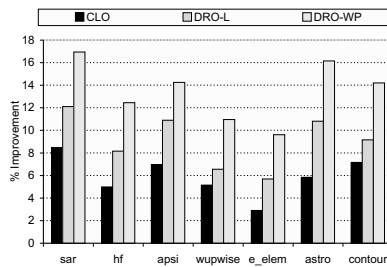


Figure 16: Performance improvements in overall execution times.

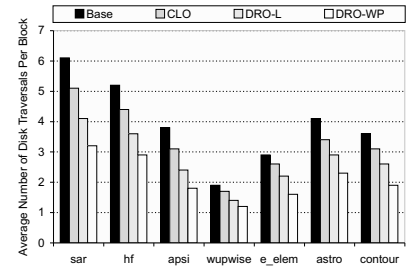


Figure 17: Average number of fetches per block. Each bar represents how many times a given block is brought to buffer cache.

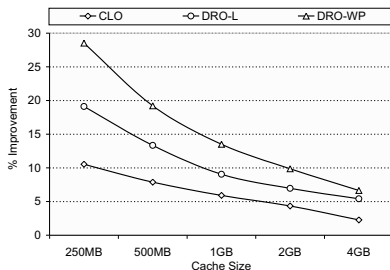


Figure 18: Sensitivity to the buffer size.

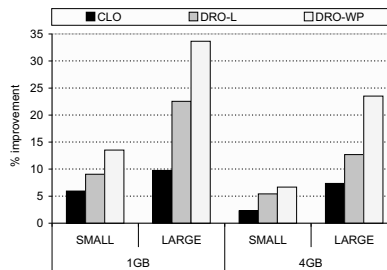


Figure 19: Sensitivity to the input size.

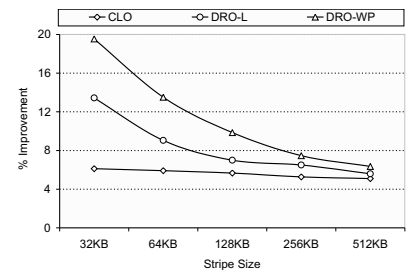


Figure 20: Sensitivity to the stripe size.

that the DRO-WP version generates the best results with all stripe sizes tested. We also see that the performance savings are higher with smaller stripe sizes. This can be explained as follows. The disk reuse is not very good in the original codes (the base scheme) with smaller stripe sizes, and since the savings shown are normalized with respect to the original codes, we observe large savings.

Comparison with I/O Prefetching. We next compare our approach to prefetching. The specific prefetch implementation we use is inspired by TIP [24], a hint-based I/O prefetching scheme. The graph in Figure 21 presents, for each benchmark, three results: prefetching alone, DRO-WP alone, and the two technique combined. We see from these results that the best performance improvements are achieved using both prefetching and code restructuring. This is because these two optimization techniques are in a sense orthogonal to each other. Specifically, while core restructuring tries to *reduce I/O latencies* by improving buffer performance, prefetching tries to *hide I/O latencies*. While it is also possible to integrate these two optimizations better (rather than applying one after another), we postpone exploring this option to a future study.

Impact of File Layout Optimization. Recall from Section 4 that file layout optimization (which impacts the layout of data on the disks as well) can help our approach improve disk reuse further. To quantify this, we

performed another set of experiments, whose results are presented in Figure 22 when the whole program is optimized. We see that, except for one benchmark, layout optimization improves the effectiveness of our code restructuring approach. The average *additional* improvement it brings is about 7%. We observe that the file layout optimization could not find much opportunity for improvement in benchmark *e_elem* as the default file layouts of the disk resident arrays in this benchmark perform very well.

Evaluation of Parallel Execution. Figure 23 presents the results collected from an evaluation of the parallel version of our approach discussed in Section 5. For these experiments, the number of CPUs that are used to execute an application is varied between 1 and 8. For each processor size, we present the results with our baseline implementation (where disk reuse is optimized from each CPU's perspective individually) as well as with those obtained when the approach in Section 5 is enabled. We see from these results that considering all parallel threads together is important in maximizing overall disk reuse, especially with the large number of CPUs. For example, when 8 CPUs are used for executing an application, the average performance improvements with individual reuse optimization (sequential version) and collective reuse optimization (parallel version) are 25.7% and 33.3%, respectively.

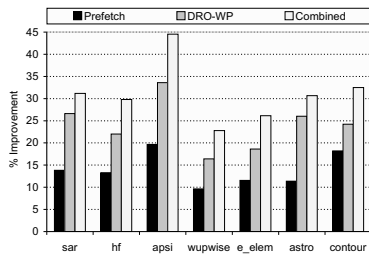


Figure 21: Comparison with I/O prefetching.

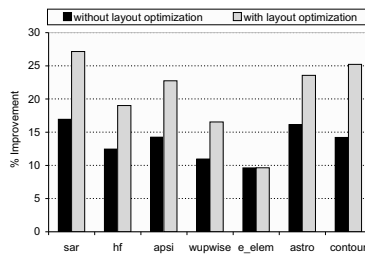


Figure 22: Impact of layout optimization.

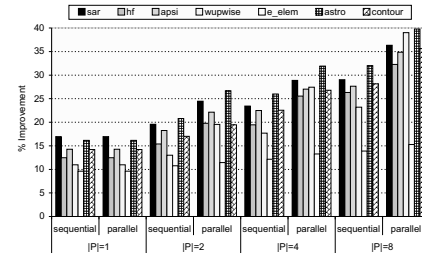


Figure 23: Impact of parallel thread optimization.

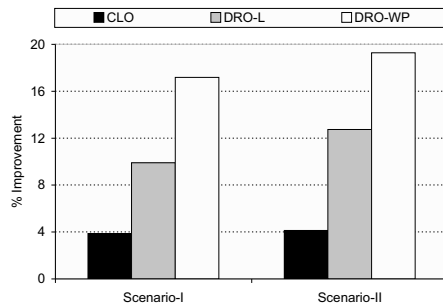


Figure 24: Impact of multiple application execution.

Evaluation of Multiple Application Execution. Since a disk system can be used by multiple applications at the same time, it is also important to quantify the benefits brought by our approach under such an execution scenario. Figure 24 presents the results from two sets of experiments. In the first set, called Scenario-I, 7 CPUs are used and each CPU executes one of our seven applications. In the second set, called Scenario-II, 8 CPUs are used and four of our applications (sar, hf, apsi and astro) are parallelized, each using two CPUs. In both set of experiments, a CPU executes only a single thread. Also, in Scenario-I the sequential version of our approach is used (for each application), and in Scenario-II the parallel version explained in Section 5 is used. The results given in Figure 24 indicate that the DRO-WP version generates the best results for both the scenarios tested.

7 Discussion of Related Work

An important way of using compiler in improving I/O performance is to hide I/O latency through I/O prefetching. Mowry et al [21] proposed a compiler-managed I/O prefetching technique for out-of-core applications. They automated the insertion of I/O prefetching instructions based on future memory page usage determined by compile-time locality analysis. In another study [2], they proposed a runtime system for managing the dynamic behavior of compiler-inserted I/O prefetch/release hints from multiple applications running concurrently. Several

researches focused on compilation of I/O-intensive applications. For example, Bordawekar et al [1] focused on stencil computations and proposed several algorithms to optimize communication and ultimately to enhance I/O performance. Paleczny et al [23] proposed a technique to guide I/O for out-of-core applications based on high-level annotations, which they incorporated into Fortran D compiler. To bridge the disparities between the data access patterns and the storage (file) layouts, [18] and [17] proposed compiler-directed I/O optimization strategies. Their main idea is to find the most preferable I/O access patterns to disk-resident files, and then determine the most suitable storage layouts associated with them. For the remaining part, which contains non-dominant access patterns, they optimized it using collective I/O operations. In [35], Vilayannur et al proposed a compiler-directed discretionary caching policies for I/O-intensive applications. They leveraged compiler support in determining the cache blocks to be accessed in each loop nest. Besides these efforts to enhance the I/O performance explained so far, the compiler-guided information can be used for different purposes, such as reducing energy consumption of disk subsystems. In [30], Son et al proposed to expose disk layouts of each data file to the compiler and let the compiler analyze the data access patterns along with this information to determine I/O (disk) access patterns. These extracted disk access patterns are finally used for transforming code and/or underlying disk layouts to reduce the energy consumption of disk subsystems.

There has been significant past work on optimizing file and buffer cache management in storage systems [3, 9, 10, 16, 28, 35]. To better exploit multi-level caches, which are common in modern storage systems, several multi-level buffer cache management policies have been proposed [6, 37, 35]. [37] introduced a DEMOTE operation that allows one to keep cache blocks in an exclusive manner, i.e., a cache block is not duplicated across cache hierarchy. Chen et al [6], on the other hand, utilize the eviction information of higher level cache in deciding which cache blocks need to be replaced in a lower level of the cache hierarchy. More recently, [38] pro-

posed a replacement policy for multi-level cache, called Karma. Karma uses application hints in maintaining cache blocks exclusively. Most high-end parallel and cluster systems provide some sort of parallel I/O operations to meet the I/O requirements (i.e., low latency and high bandwidth) of scientific applications. This is typically accomplished by employing a set of I/O nodes, each of which is equipped with multiple disks, dividing a file into a number of small file stripes, and distributing those stripes across available I/O nodes. This notion of file-level striping is adapted in many commercial or research parallel file systems, such as IBM GPFS [28], Intel PFS [11], PPFS [15], Galley [22], and PVFS2 [27]. It should be mentioned that these parallel file systems provide huge I/O performance improvements when they receive large and contiguous I/O requests. However, many scientific applications that exhibit small and non-contiguous I/O access patterns may suffer from performance degradation. To deal with this problem, several approaches have been proposed in the context of different parallel file system libraries and APIs such as Panda [5], PASSION [31], and MPI-IO [32, 33, 34]. Among various techniques used to achieve this goal, collective I/O is commonly recognized as an efficient way of reducing I/O latency. The concept of collective I/O can be implemented in different places of parallel I/O systems; namely, client side [34], disk side [19], or server side [29]. The majority of the existing collective I/O implementations employ two-phase I/O [34]. In two-phase I/O, disk accesses are reorganized in client side (compute node) before sending them over the I/O nodes. Disk-directed I/O [19], on the other hand, performs collective I/O operations on disk side, where I/O requests are optimized such that they conform to the storage layouts. In Panda [29], I/O server nodes, rather than disk or client nodes, generate I/O requests that conform to the layouts of disk-resident array data.

Our approach is different from prior compiler-based I/O optimization techniques in that it optimizes entire program rather than individual, parallel loop nests. It is also different from previous studies that considered buffer caching and prefetching because we improve I/O performance by increasing disk reuse. Lastly, our approach emphasizes the role of file layout optimization and parallel execution when applying optimization techniques to achieve better disk reuse (and better cache performance).

8 Concluding Remarks and Future Work

In the recent past, sensor, measurement, and simulation-based applications in science and engineering have grown dramatically in complexity. Moreover, there have been huge increases in the sizes of the data sets they pro-

duce, manipulate, and consume, meaning that the high I/O performance is a must for these applications. Unfortunately, advances in I/O architectures (in particular, disks) could not meet this high I/O performance requirement satisfactorily. As a result, adequate software support for I/O is critical and has to be provided at different layers, including libraries, file systems, runtime systems, and compilers. The main contribution of this paper is a compiler-directed disk performance optimization scheme for large-scale data-intensive applications. This proposed scheme is oriented towards maximizing disk reuse over successive visits to the disk system within a given period of time, thereby (1) maximizing the utilization of cache in the computation node, and (2) reducing the latencies due to data search on disks. In addition, the success of this scheme can be increased significantly if it is augmented with a file layout optimization scheme, and it can be easily adapted to capture disk interactions across the threads of a parallel application. We implemented this scheme fully using an optimizing compiler framework and evaluated its performance using seven data-intensive applications that exercise disks. The results collected indicate that our compiler-directed approach is very successful in maximizing disk reuse, and this in turn results in large savings in I/O latencies. In our experiments, we also compared our approach to a conventional data reuse optimization scheme (not designed for I/O) and explain where the additional benefits are coming from. This work shows how an optimizing compiler can help reduce I/O latencies by automated code restructuring. We believe that further compiler optimizations are possible by exposing the disk layout of data to the different layers of the software stack. One of the research directions to investigate is this interaction between the compiler optimizations for I/O and other I/O optimizations that are normally applied by file systems and runtime libraries. Another interesting research direction is to adapting the cache policy based on the application behavior information collected by the compiler. This can help to increase the hit rates of the cache, thereby further boosting the performance of the application.

Acknowledgments This work is supported in part by the NSF grants #0406340, #0444158, and #0621402. We would like to thank our anonymous reviewers for their helpful comments.

References

- [1] BORDAWEKAR, R., CHOUDHARY, A. N., AND RAMANUJAM, J. Automatic Optimization of Communication in Compiling Out-of-Core Stencil Codes. In *International Conference on Supercomputing* (1996), pp. 366–373.
- [2] BROWN, A. D., AND MOWRY, T. C. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory

- Intelligently. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation* (2000), pp. 31–44.
- [3] CAO, P., FELTEN, E. W., AND LI, K. Application-Controlled File Caching Policies. In *Proceedings of the USENIX Technical Conference* (1994), pp. 11–11.
- [4] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: High-Performance, Reliable Secondary Storage. *ACM Comput. Surv.* 26, 2 (1994), 145–185.
- [5] CHEN, Y., WINSLETT, M., CHO, Y., AND KUO, S. Automatic Parallel I/O Performance Optimization in Panda. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures* (1998), pp. 108–118.
- [6] CHEN, Z., ZHOU, Y., AND LI, K. Eviction-based Cache Placement for Storage Caches. In *USENIX Annual Technical Conference* (2003), pp. 269–281.
- [7] CHENG CAI, M., DENG, X., AND ZANG, W. A TDI System and Its Application to Approximation Algorithms. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science* (1998), pp. 227–231.
- [8] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.
- [9] CORTES, T., GIRONA, S., AND LABARTA, J. Design Issues of a Cooperative Cache with No Coherence Problems. In *Proceedings of the 5th Workshop on I/O in Parallel and Distributed Systems* (1997), pp. 37–46.
- [10] FORNEY, B. C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002).
- [11] GARG, S. TFLOPS PFS: Architecture and Design of a Highly Efficient Parallel File System. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1998), pp. 1–12.
- [12] GROPP, W., THAKUR, R., AND LUSK, E. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.
- [13] HALL, M. W., ANDERSON, J. M., AMARASINGHE, S. P., MURPHY, B. R., LIAO, S.-W., BUGNION, E., AND LAM, M. S. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer* 29, 12 (1996), 84–89.
- [14] HENNING, J. L. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer* 33, 7 (2000), 28–35.
- [15] JAMES V. HUBER, J., CHIEN, A. A., ELFORD, C. L., BLUMENTHAL, D. S., AND REED, D. A. PPFS: a high performance portable parallel file system. In *Proceedings of the 9th International Conference on Supercomputing* (1995), pp. 385–394.
- [16] KALLAHALLA, M., AND VARMAN, P. J. Optimal Prefetching and Caching for Parallel I/O Systems. In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures* (2001), pp. 219–228.
- [17] KANDEMIR, M. A Collective I/O Scheme Based on Compiler Analysis. In *Proceedings of the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (2000), pp. 1–15.
- [18] KANDEMIR, M., AND CHOUDHARY, A. Compiler-Directed I/O Optimization. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing* (2002), p. 19.2.
- [19] KOTZ, D. Disk-directed I/O for an Out-of-core Computation. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (1995), pp. 159–166.
- [20] MOORE, F. Disk companies pricing themselves out of business again: lessons of the past still unlearned. *Computer Technology Review* (March 2003).
- [21] MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (1996), pp. 3–17.
- [22] NIEUWEJAAR, N., AND KOTZ, D. The Galley Parallel File System. In *Proceedings of the 10th International Conference on Supercomputing* (1996), pp. 374–381.
- [23] PALECZNY, M., KENNEDY, K., AND KOELBEL, C. Compiler Support for Out-of-Core Arrays on Data Parallel Machines. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation* (McLean, VA, 1995), pp. 110–118.
- [24] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), pp. 79–95.
- [25] PINHEIRO, E., BIANCHINI, R., AND DUBNICKI, C. Exploiting Redundancy to Conserve Energy in Storage Systems. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (2006), pp. 15–26.
- [26] PUGH, W. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1991), pp. 4–13.
- [27] PVFS2 DEVELOPMENT TEAM. Parallel Virtual File System, Version 2. <http://www.pvfs.org/pvfs2-guide.html>, September 2003.
- [28] SCHMUCK, F., AND HASKIN, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st Conference on File and Storage Technologies* (January 2002), pp. 231–244.
- [29] SEAMONS, K. E., CHEN, Y., JONES, P., JOZWIAK, J., AND WINSLETT, M. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing* (1995).
- [30] SON, S. W., KANDEMIR, M., AND CHOUDHARY, A. Software-Directed Disk Power Management for Scientific Applications. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium* (2005).
- [31] THAKUR, R., BORDAWEKAR, R., CHOUDHARY, A., PONNUSAMY, R., AND SINGH, T. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference* (1994), pp. 119–128.
- [32] THAKUR, R., GROPP, W., AND LUSK, E. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation* (1999), pp. 182–189.
- [33] THAKUR, R., GROPP, W., AND LUSK, E. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems* (1999), pp. 23–32.
- [34] THAKUR, R., GROPP, W., AND LUSK, E. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing* 28, 1 (2002), 83–105.
- [35] VILAYANNUR, M., SIVASUBRAMANIAM, A., KANDEMIR, M. T., THAKUR, R., AND ROSS, R. B. Discretionary Caching for I/O on Clusters. In *IEEE International Symposium on Cluster Computing and the Grid* (2003), pp. 96–103.
- [36] WOLFE, M. J. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [37] WONG, T. M., AND WILKES, J. My Cache or Yours? Making Storage More Exclusive. In *USENIX Annual Technical Conference* (2002), pp. 161–175.
- [38] YADGAR, G., FACTOR, M., AND SCHUSTER, A. Karma: Know-it-All Replacement for a Multilevel Cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007), pp. 25–25.
- [39] ZHANG, Y. Z., KUNG, E. H., AND HAWORTH, D. C. A PDF Method for Multidimensional Modeling of HCCI Engine Combustion: Effects of Turbulence/Chemistry Interactions on Ignition Timing and Emissions. In *Proceedings of the 30th International Symposium on Combustion* (2004), pp. 2763–2771.
- [40] ZHU, Q., AND ZHOU, Y. Power-Aware Storage Cache Management. *IEEE Trans. Comput.* 54, 5 (2005), 587–602.

Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems

Amar Phanishayee, Elie Krevat, Vijay Vasudevan,
David G. Andersen, Gregory R. Ganger, Garth A. Gibson, Srinivasan Seshan

Carnegie Mellon University

Abstract

Cluster-based and iSCSI-based storage systems rely on standard TCP/IP-over-Ethernet for client access to data. Unfortunately, when data is striped over multiple networked storage nodes, a client can experience a TCP throughput collapse that results in much lower read bandwidth than should be provided by the available network links. Conceptually, this problem arises because the client simultaneously reads fragments of a data block from multiple sources that together send enough data to overload the switch buffers on the client's link. This paper analyzes this *Incast* problem, explores its sensitivity to various system parameters, and examines the effectiveness of alternative TCP- and Ethernet-level strategies in mitigating the TCP throughput collapse.

1 Introduction

Cluster-based storage systems are becoming an increasingly important target for both research and industry [1, 36, 15, 24, 14, 8]. These storage systems consist of a networked set of smaller storage servers, with data spread across these servers to increase performance and reliability. Building these systems using commodity TCP/IP and Ethernet networks is attractive because of their low cost and ease-of-use, and because of the desire to share the bandwidth of a storage cluster over multiple compute clusters, visualization systems, and personal machines. Furthermore, non-IP storage networking lacks some of the mature capabilities and breadth of services available in IP networks. However, building storage systems on TCP/IP and Ethernet poses several challenges. In this paper, we analyze one important barrier to high-performance storage over TCP/IP: the *Incast* problem [24].¹

Incast is a catastrophic TCP throughput collapse that occurs as the number of storage servers sending data to a

client increases past the ability of an Ethernet switch to buffer packets. As we explore further in §2, the problem arises from a subtle interaction between limited Ethernet switch buffer sizes, the communication patterns common in cluster-based storage systems, and TCP's loss recovery mechanisms. Briefly put, data striping couples the behavior of multiple storage servers, so the system is limited by the request completion time of the *slowest* storage node [7]. Small Ethernet buffers are exhausted by a concurrent flood of traffic from many servers, which results in packet loss and one or more TCP timeouts. These timeouts impose a delay of hundreds of milliseconds—orders of magnitude greater than typical data fetch times—significantly degrading overall throughput.

This paper provides three contributions. First, we explore in detail the root causes of the *Incast* problem, characterizing its behavior under a variety of conditions (buffer space, varying numbers of servers, etc.). We find that *Incast* is a general barrier to increasing the number of source nodes in a cluster-based storage system. While increasing the amount of buffer space available can delay the onset of *Incast*, any particular switch configuration *will* have some maximum number of servers that can send simultaneously before throughput collapse occurs.

Second, we examine the effectiveness of existing TCP variants (e.g., Reno [3], NewReno [13], SACK [22], and limited transmit [2]) designed to improve the robustness of TCP's loss recovery. While we do find that the move from Reno to NewReno substantially improves performance, none of the additional improvements help. Fundamentally, when TCP loses *all* packets in its window or loses retransmissions, no clever loss recovery algorithms can help.

Third, we examine a set of techniques that are moderately effective in masking *Incast*, such as drastically reducing TCP's retransmission timeout timer. With some of these solutions, building a high-performance, scalable cluster storage system atop TCP/IP and Ethernet can be practical. Unfortunately, while these techniques can be effective, none of them is without drawbacks. Our final

¹Some people use the term *incast* to characterize many-to-one communication. In this paper, we use the term *Incast* to refer to TCP throughput collapse in a *synchronized reads* setting.

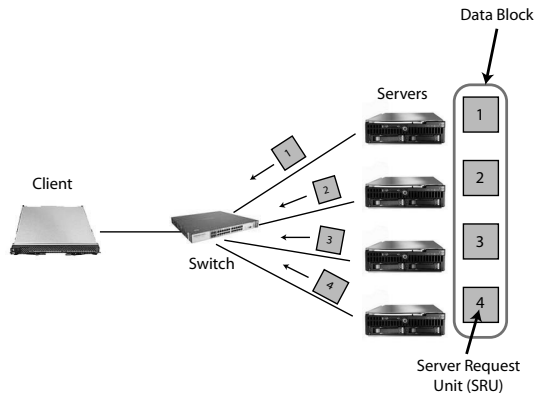


Figure 1: A simple cluster-based storage environment with one client requesting data from multiple servers through synchronized reads.

conclusion is that no existing solutions are entirely sufficient, and further research is clearly indicated to devise a principled solution for the *Incast* problem.

2 Background

In cluster-based storage systems, data is stored across many storage servers to improve both reliability and performance. Typically, their networks have high bandwidth (1-10 Gbps) and low latency (round trip times of tens to hundreds of microseconds) with clients separated from storage servers by one or more switches.

In this environment, data blocks are striped over a number of servers, such that each server stores a fragment of a data block, denoted as a *Server Request Unit (SRU)*, as shown in Figure 1. A client requesting a data block sends request packets to all of the storage servers containing data for that particular block; the client requests the next block only after it has received all the data for the current block. We refer to such reads as *synchronized reads*.

This simple environment abstracts away many details of real storage systems, such as multiple stripes per data block, multiple outstanding block requests from a client, and multiple clients on a single switch making requests across a shared subset of servers. However, this is the most basic representative setting in which *Incast* can occur and simplifies our analysis.

The need for a high performance environment that supports parallel operations such as synchronized reads is particularly important because of such recent projects as *Parallel NFS (pNFS)*. *pNFS* is a component of *NFSv4.1* that supports parallel data transfers and data striping across multiple file servers [37, 28, 18].

Most networks are provisioned so the client's link capacity to the switch is the throughput bottleneck of any

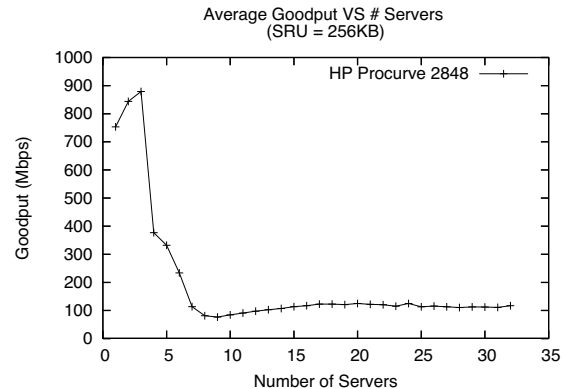


Figure 2: TCP throughput collapse for a synchronized reads application performed on a storage cluster.

parallel data transfer [16, 21]. Unfortunately, when performing synchronized reads for data blocks across an increasing number of servers, a client may observe a TCP throughput drop of one or two orders of magnitude below its link capacity. Figure 2 illustrates this performance drop in a cluster-based storage network environment when a client requests data from just seven servers.

Early parallel network storage projects, such as the NASD project [15], observed TCP throughput collapse in cluster-based storage systems during synchronous data transfers. This was documented as part of a larger paper by Nagle et al. [24], who termed the problem *Incast* and attributed it to multiple senders overwhelming a fixed-size switch buffer. However, while Nagle demonstrated the problem and suggested that an alternative TCP implementation shows a modest improvement, a full analysis and measurement of the problem was not performed nor were possible solutions presented.

Incast has not been thoroughly studied. Current systems attempt to avoid TCP throughput collapse by limiting the number of servers involved in any block transfer, or by artificially limiting the rate at which they transfer data. These solutions, however, are typically specific to one configuration (e.g. a number of servers, data block sizes, link capacities, etc.), and thus are not robust to changes in the storage network environment.

3 Experimental Setup

In this section, we describe the simulation and real system environments where we measure the effects of *Incast* and the corresponding workloads that we use in both settings.

Parameter	Default
Number of servers	—
<i>SRU</i> Size	256KB
Link Bandwidth	1 Gbps
Round Trip Time (RTT)	100 μ s
Per-port switch output buffer size	—
TCP Implementation: Reno, NewReno, SACK	NewReno
Limited Transmit	disabled
Duplicate-ACK threshold (da_{thresh})	3
Slow Start	enabled
RTO_{min}	200ms

Table 1: Simulation parameters with default settings.

3.1 Simulation Environment

All of our simulations use *ns-2* [27], an event-driven network simulator that models networked applications at the packet granularity. Our default simulation configuration consists of one client and multiple servers all connected to the same switch as shown in Figure 1.

Table 1 shows the parameters and their corresponding default values that we vary in simulation. We choose a 256KB default *SRU* size to model a production storage system [9]. From our simulations, we obtain global and per-flow TCP statistics such as retransmission events, timeout events, TCP window sizes, and other TCP parameters to aid in our analysis of *Incast*.

Our test application performs synchronized reads over TCP in *ns-2* to model a typical striped file system data transfer operation. The client requests a data block from n servers by sending a request packet to each server for one *SRU* worth of data. When a client receives the entire data block of $n \cdot SRU$ total bytes, it immediately sends request packets for the next block. Each measurement runs for 20 seconds of simulated time, transferring enough data to accurately calculate throughput.

3.2 Cluster-based Storage Environment

Our experiments use a networked group of storage servers as configured in production storage systems. Our application performs the same synchronized reads protocol as in simulation and measures the achieved throughput. All systems have 1 Gbps links and a client-to-server Round Trip Time (RTT) of approximately 100 μ s. We evaluated three different storage clusters:

- **Procurve:** One or more HP Procurve 2848 Ethernet switches configured in a tree hierarchy connect a client to up to 64 servers, each running Linux 2.6.18 SMP.²
- **S50:** A Force10 S50 switch connects 48 Redhat4 Linux 2.6.9-22 machines on one switch (1 client, 47

²Although this topology does not exactly match our simulation topology, we find that multiple switches do not prevent *Incast*.

servers).

- **E1200:** A Force10 E1200 switch with 672 ports with at least 1MB output buffer per port. This switch connects 88 Redhat4 Linux 2.6.9-22 machines (1 client, 87 servers).

For our workload and analysis, we keep the *SRU* size fixed while we scale the number of servers, implicitly increasing the data block size with the number of servers.³

4 Reproducing Incast

In this section, we first demonstrate *Incast* occurring in several real-world cluster-based storage environments. Using simulation, we then show that *Incast* is a generic problem and identify the causes of *Incast*. We find that the results obtained from our experimental setup validate our simulation results. Finally, we show that attempts to mitigate *Incast* by varying parameters such as switch buffer size and *SRU* size are incomplete solutions that either scale poorly or introduce system inefficiencies when interacting with a storage system.

4.1 Incast in real systems

To ensure that the throughput collapse shown in Figure 2 is not an isolated instance, we study *Incast* on the three storage clusters described in §3.2. Figure 3 indicates that both the Procurve and S50 environments experience up to an order of magnitude drop in *goodput* (throughput as observed by the application). The E1200, however, did not exhibit any throughput drop for up to the 87 available servers, which we attribute to the large amount of buffer space available on the switch.

In our analysis, we use estimates of the output buffer sizes gathered from network administrators and switch specifications. Unfortunately, we are unable to determine the exact per-port buffer sizes on these switches.⁴ This information is not available because most switches dynamically allocate each link's output buffer from a shared memory pool. Also, when QoS queues are enabled, the amount of memory allocated to the queues depends on vendor-specific implementations. However, our estimates for output buffer sizes are corroborated by simulation results.

³Some storage systems might instead scale by keeping the block size fixed and increasing the number of servers used to stripe data over. We explore the effects of a fixed block size scaling model in §4.

⁴While most modern switches use Combined Input-Output Queuing (CIOQ), we focus our attention on the output buffer size. Since our simulation results are validated by our experimental results, we did not find it necessary to model additional complexity into our switches in simulation.

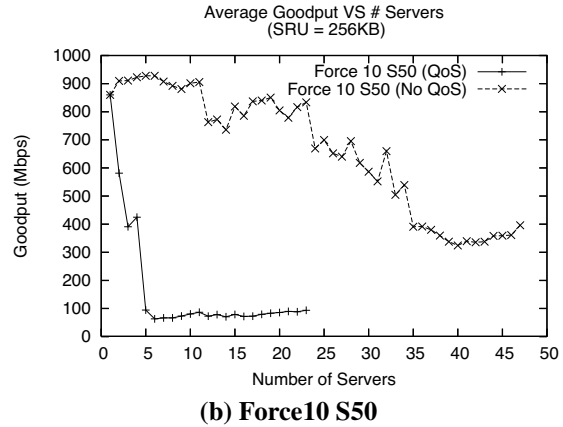
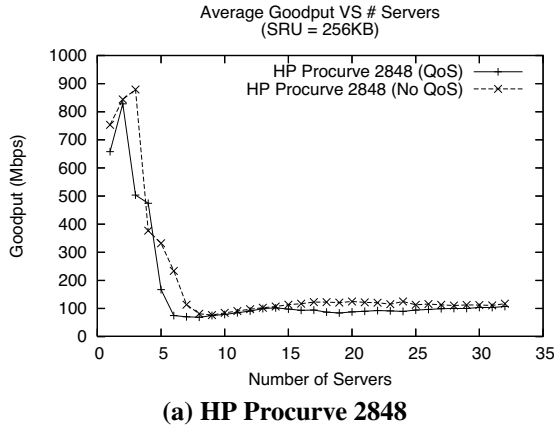


Figure 3: *Incast* observed on different switch configurations with and without QoS support. Disabling QoS support has only a small effect for (a) the HP Procurve 2848, but significantly delays the onset of *Incast* for (b) the Force10 S50.

Many switches provide QoS support to enable prioritization of different kinds of traffic. A common implementation technique for providing QoS is to partition the output queue for each class of service. As a result, disabling QoS increases the effective size of the output queues, though the amount of this increase varies across different vendors and switches. As shown in Figure 3(a), disabling QoS on the Procurve environment does not significantly affect throughput – a collapse still occurs around 7 servers. In contrast, Figure 3(b) shows that disabling QoS on the Force10 S50 significantly delays the onset of *Incast*. These results suggest that the Force10 S50 allocates a relatively larger amount of buffer space and switch resources to QoS support in comparison to the Procurve 2848. Switch buffer sizes play an important role in mitigating *Incast*, as we evaluate in §4.3.

4.2 Validation and Analysis in Simulation

To determine how general a problem *Incast* is for cluster-based storage over TCP/IP/Ethernet, we also reproduce *Incast* in the *ns-2* network simulator. Figure 4 shows *Incast* in simulation with an order of magnitude collapse at around 8 servers and beyond. These results closely match those from the Procurve environment. The differences between the results, including the difference in behavior below 3 servers, have a few possible causes. First, simulated source nodes serve data as rapidly as the network can handle, while real systems often have other slight delays. We attribute the lower performance of the real system between 1-3 servers to these differences. Also, simulation does not model Ethernet switching behavior, which may introduce small timing and performance differences.

Despite these differences, real world experiments validate our simulation measurements, showing that the im-

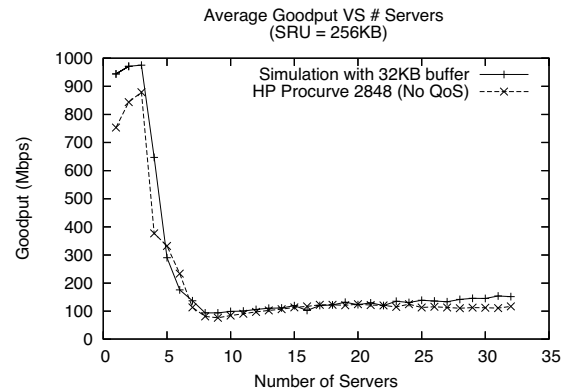


Figure 4: Comparison of *Incast* in simulation and in real world cluster-based settings.

pact of *Incast* is nearly identical in both real world system measurements and simulation.

An analysis of the TCP traces obtained from simulation reveals that TCP retransmission timeouts are the primary cause of *Incast* (Figure 5).⁵ When goodput degrades, most servers still send their *SRU* quickly, but one or more other servers experience a timeout due to packet losses. The servers that finish their transfer do not receive the next request from the client until the client receives the complete data block, resulting in an underutilized link.

⁵TCP goodput could also be degraded by a large number of packet retransmissions that waste network capacity. We find, however, that retransmitted packets make up only about 2% of all transmissions. This overhead is not significant when compared to the penalty of a retransmission timeout.

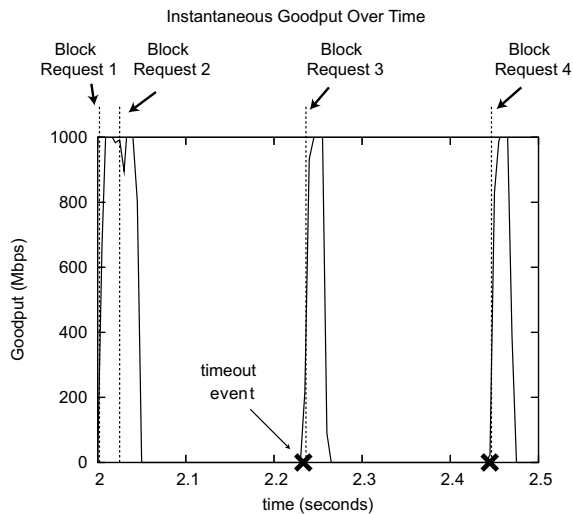


Figure 5: Instantaneous goodput averaged over 5ms intervals. Timeouts are the primary cause of *Incast*, and one stalled flow during a block transfer results in an idle link duration of around 200ms. Timeout events indicate when a flow begins recovery.

4.2.1 Why Timeouts Occur

The rest of the paper assumes a familiarity with TCP terms and concepts. For a brief refresher on TCP, we refer the reader to the Appendix.

Reading blocks of data results in simultaneous transmission of packets from servers. Because the buffer space associated with the output port of the switch is limited, these simultaneous transmissions can overload the buffer resulting in losses. TCP recovers from losses by retransmitting packets that it has detected as being lost. This loss detection is either data-driven or is based on a timeout for a packet at the sender.

A TCP sender assigns sequence numbers to transmitted packets and expects TCP acknowledgements (ACKs) for individual packets from the receiver. The TCP receiver acknowledges the last packet it received in-order. Out-of-order packets generate duplicate ACKs for the last packet received in-order. Receiving multiple duplicate ACKs for a packet is an indication of a loss – this is data-driven loss detection. Timeouts are used as a fallback option in the absence of enough feedback, and are typically an indication of severe congestion.

In Figure 3(a), we see an initial drop from 900Mbps to 500Mbps between 3-5 servers on the Procurve. Analysis of TCP logs reveal that this drop in throughput is caused by the delayed ACK mechanism [3]. In the delayed ACK specification, an acknowledgement should be generated for at least every second packet and must be generated within 200ms of the arrival of the first unacknowledged packet. Most TCP implementations wait only

Finding	Location
<i>Incast</i> is caused by too-small switch output buffers: increasing buffer size can alleviate the situation.	§4.3
TCP NewReno and SACK improve goodput considerably over TCP Reno, but do not prevent <i>Incast</i> .	§5.1.1
Improvements to TCP loss recovery using Limited Transmit or reducing the Duplicate ACK threshold do not help.	§5.1.2
Reducing the penalty of a timeout by lowering the minimum retransmission value can help significantly, but poses questions of safety and generality.	§5.2
Enabling Ethernet Flow Control is effective only in the very simplest setting, but not for more common multi-switched systems.	§6

Table 2: Summary of Major Results.

40ms before generating this ACK. This 40ms delay causes a “mini-timeout”, leading to underutilized link capacity similar to a normal timeout. However, normal timeouts are responsible for the order of magnitude collapse seen beyond 5 servers in *Incast*. We explore TCP-level solutions to avoid timeouts and to reduce the penalty of timeouts in detail in §5.

4.3 Reducing Losses: Larger Switch Buffers

Since timeouts are the primary cause of *Incast*, we try to prevent the root cause of timeouts – packet losses – to mitigate *Incast* by increasing the available buffer space allocated at the Ethernet switch. §4.1 mentioned that a larger switch buffer size delays the onset of *Incast*. Figure 6 shows that doubling the size of the switch’s output port buffer in simulation doubles the number of servers that can transmit before the system experiences *Incast*.

With a large enough buffer space, *Incast* can be avoided for a certain number of servers, as shown in Figure 6. This is corroborated by the fact that we were unable to observe *Incast* with 87 servers on the Force10 E1200 switch, which has very large buffers. But Figure 6 shows that for a 1024KB buffer, 64 servers only utilize about 65% of the client’s link bandwidth, and doubling the number of servers only improves goodput to 800Mbps.

Unfortunately, switches with larger buffers tend to cost more (the E1200 switch costs over \$500,000 USD), forcing system designers to choose between overprovisioning, future scalability, and hardware budgets. Furthermore, switch manufacturers may need to move to faster and more expensive memory (e.g., SRAM) as they

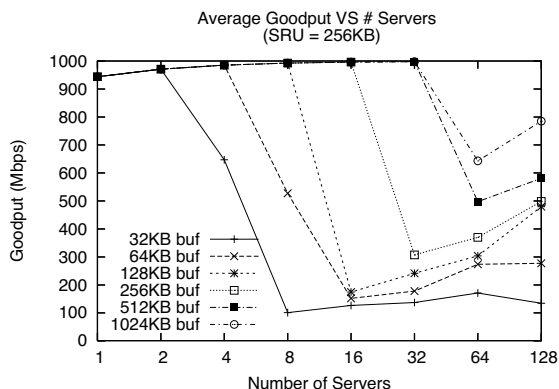


Figure 6: Effect of varying switch buffer size – doubling the size of the switch’s output port buffer doubles the number of servers that can be supported before the system experiences *Incast*.

move to 10Gbps and beyond. This move places cost pressure on manufacturers to keep buffer sizes small. Hence a more cost-effective solution other than increasing buffer sizes is required.

While the above experiments are run in a controlled environment with only one client reading from many servers, real storage environments are likely more complicated, with many clients making multiple concurrent requests to different sets of servers. Since the amount of buffer space available per client request likely decreases in common shared memory switch architectures (and does not increase otherwise), we expect overall performance to be worse in these more complex environments.

4.4 Reducing Idle Link Time by Increasing SRU Size

Figure 7 illustrates that increasing the *SRU* size improves the overall goodput. With 64 servers, the 1000KB *SRU* size run is two orders of magnitude faster than the 10KB *SRU* size run. Figure 8 shows that real switches, in this case the Force10 S50, behave similarly.

TCP performs well in settings without synchronized reads, which can be modeled by an infinite *SRU* size. The simple TCP throughput tests in *netperf* do not exhibit *Incast* [24]. With larger *SRU* sizes, servers will use the spare link capacity made available by any stalled flow waiting for a timeout event; this effectively reduces the ratio of timeout time to transfer time.

A large *SRU* size helps maximize disk head utilization on reads. Unfortunately, an *SRU* size of even 8MB is quite impractical: most applications ask for data in small chunks, corresponding to an *SRU* size range of 1-256KB. For example, when requesting an 8MB block from the storage system, one would like to stripe this block across

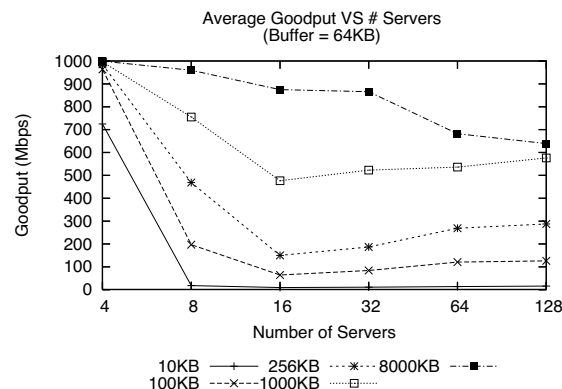


Figure 7: Effect of varying *SRU* size – for a given number of servers, a larger *SRU* improves goodput.

as many servers as needed to saturate the link. In addition, a larger *SRU* size can increase lock contention due to overlapping writes, leading to poor write performance in file system implementations [9].

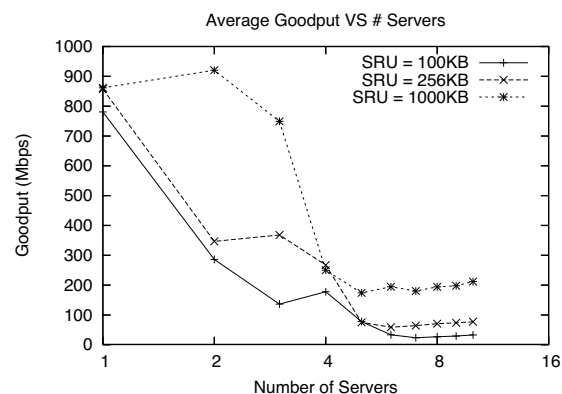


Figure 8: Effect of varying *SRU* size for Force10 S50 with QoS support enabled.

Figure 9 shows an alternative scaling model in simulation where the data block size is fixed and the number of storage servers are increased, placing an upper bound per request on the amount of pinned kernel memory in the client. This scaling model more closely resembles how file systems request data. A rapid drop-off in goodput is observed for a fixed block size as the number of servers increases. Because the data block size is fixed, increasing the number of servers reduces the *SRU* size. Thus, the effect of increasing the number of servers is compounded by a reduced *SRU* size and results in even lower goodput.

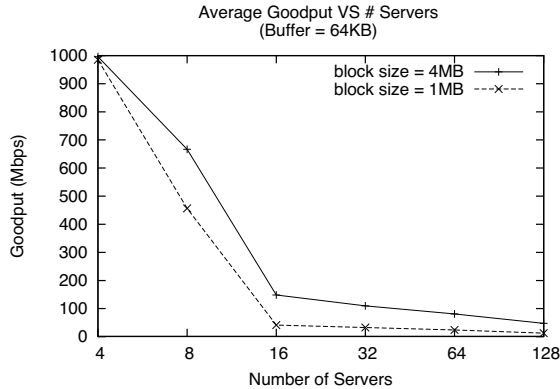


Figure 9: For a fixed data block size, as opposed to a fixed *SRU* size, increasing the number of servers also reduces the *SRU* size requested from each server and results in even lower goodput.

5 TCP-level Solutions

Because TCP timeouts are the primary reason that *In-cast* hurts throughput, we analyze TCP-level solutions designed to reduce both the number and penalty of timeouts. We perform this analysis using *ns-2* simulations.

5.1 Avoiding Timeouts

In this section, we analyze three different approaches to avoiding timeouts by:

- Improving TCP's resilience to common loss patterns by using alternative TCP implementations;
- Addressing the lack of sufficient data-driven feedback;
- Reducing the traffic injection rate of exponentially growing TCP windows during *slow-start* [3].

Analysis Method - Performance and Timeout Categorization: For each approach, we ask two questions: 1) how much does the approach improve goodput and 2) if timeouts still occur, why? To answer the second question, we look at the number of Duplicate ACKs Received at the point when a flow experiences a Timeout (the *DART* count). The purpose of this analysis is to categorize the situations under which timeouts occur to understand whether the timeout could have been avoided.

There are three types of timeouts that cannot be avoided by most TCP implementations. The first occurs when an entire window of data is lost and there is *no* feedback available for TCP to use in recovery, leading to a *DART* value of zero. We categorize this kind of timeout as a *Full Window Loss*.

The second type occurs when the last packet of an *SRU* is dropped and there is no further data available in this

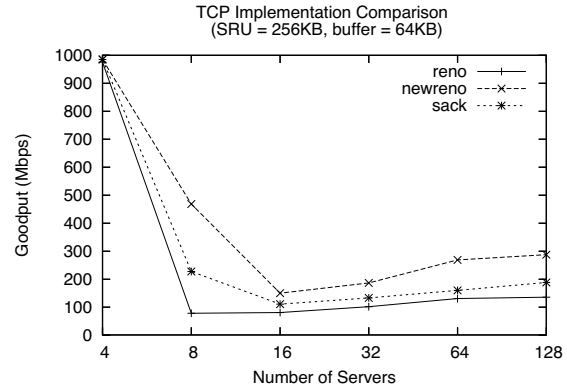


Figure 10: NewReno outperforms Reno and SACK

block request for data-driven recovery. We categorize this type of timeout as a *Last Packet Loss* case. We find, however, that there are relatively few *Last Packet Loss* cases.

The last unavoidable timeout situation occurs when a retransmitted packet triggered by TCP's loss recovery mechanism is *also* dropped. Since there is no way for the sender to know whether this retransmitted packet is dropped, the sender experiences a timeout before retransmitting the packet again. We categorize this unavoidable timeout as a *Lost Retransmit*. The *DART* count does not help in categorizing *Lost Retransmit* cases; we examine the TCP trace files to identify these situations.

5.1.1 Alternative TCP Implementations – Reno, NewReno, and SACK

Many TCP variants help reduce expensive timeouts by using acknowledgements to more precisely identify packet losses [19, 3, 13, 22]. A well-documented problem with the classic TCP Reno algorithm is that it recovers poorly from multiple losses in a window, leaving it susceptible to patterns of loss that cause a timeout [13]. For example, with a window size of six, Reno will *always* experience a timeout when the first two packets of the window are lost.

The most popular solutions to this problem are the improved retransmission algorithms in TCP NewReno [13] and the selective acknowledgements scheme in TCP SACK [22]. TCP NewReno, unlike Reno, does not exit fast recovery and fast retransmit when it receives a partial ACK (an indication of another loss in the original window), but instead immediately transmits the next packet indicated by the partial ACK. TCP SACK uses a selective acknowledgment scheme to indicate the specific packets in a window that need to be resent [12].

Figure 10 shows that both TCP NewReno and TCP SACK outperform TCP Reno. Note that TCP NewReno offers up to an order of magnitude better performance

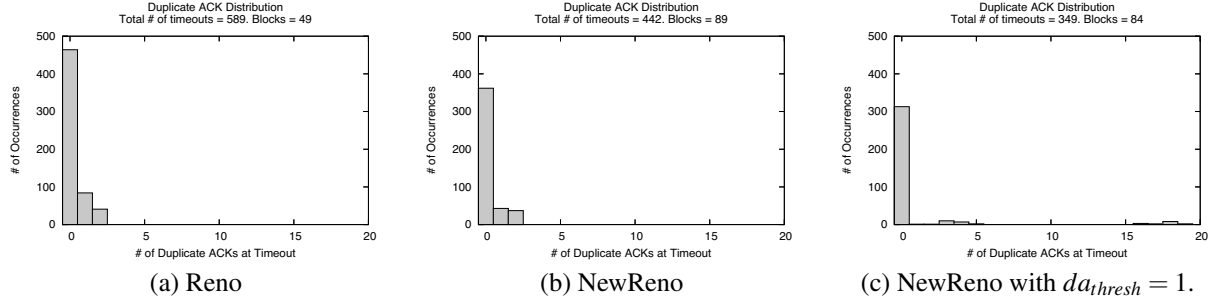


Figure 11: Distribution of Duplicate Acknowledgements Received at a Timeout (DART) recorded for a 20s run with 16 servers, 64 packet switch buffer, 256KBytes SRU size.

	Reno (Fig. 11(a))	NewReno (Fig. 11(b))	NewReno + $da_{thresh} = 1$ (Fig. 11(c))
data blocks transmitted	49	89	84
timeout events	589	442	349
full window losses	464	362	313
lost retransmits	61	2	41
lost retransmits when $DART \geq da_{thresh}$	0	0	34
lost retransmits when $DART < da_{thresh}$	61	2	7
last packets dropped	2	5	2

Table 3: Categorization of timeout events under different TCP scenarios (corresponding to Figure 11)

compared to TCP Reno in this example. Unfortunately, none of the TCP implementations can eliminate the large penalty to goodput caused by *Incast*.

Figure 11(a) and (b) shows the *DART* distribution for TCP Reno and NewReno, while Table 3 shows the categorization of timeout events. The total number of timeouts per data block is much lower for NewReno, partially explaining the goodput improvement over Reno. While most timeouts can be categorized as *Full Window Loss* cases or *Lost Retransmit* cases, there are still 78 timeouts that do not fall into these cases: they occur when the flows obtain *some*, but not enough feedback to trigger data-driven loss recovery. We next examine two schemes designed to improve these remaining cases.

5.1.2 Addressing the Lack of Sufficient Feedback – Limited Transmit and Reduced Duplicate ACK Threshold

When a flow has a small window or when a sufficiently large number of packets in a large window are lost, Limited Transmit [2] attempts to ensure that enough packets are sent to trigger the 3 duplicate ACKs necessary for data-driven recovery. Alternatively, we can reduce the duplicate ACK threshold (da_{thresh}) from 3 to 1 to automatically trigger fast retransmit and fast recovery upon receiving any duplicate acknowledgement.

Figure 12 illustrates that neither of these mechanisms provide *any* throughput benefit over TCP NewReno. We

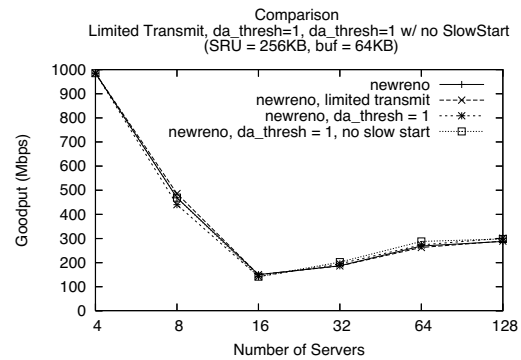
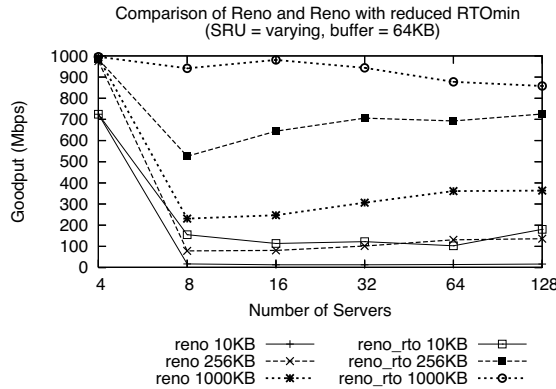
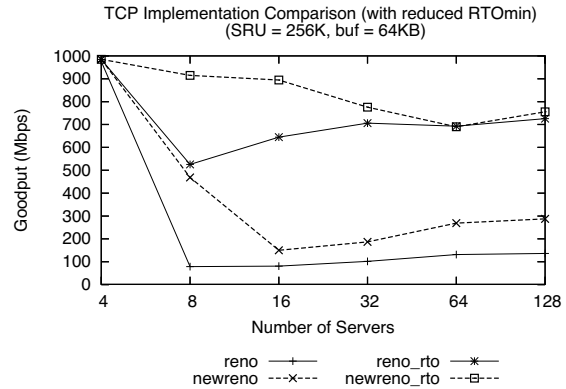


Figure 12: NewReno variants designed to improve loss recovery provide no benefit.

plot the *DART* distribution for setting $da_{thresh}=1$ in Figure 11(c). The reduced retransmit variant successfully eliminates timeouts when only 1 or 2 duplicate ACKs were received. Unfortunately, this improvement does not increase goodput because each data block transfer still experiences at least one timeout. These remaining timeouts are mostly due to full window losses or lost retransmissions, which none of the TCP variants we study can eliminate.



(a) Varying SRU sizes



(b) Different TCP implementations

Figure 13: A lower RTO value ($RTO_{min} = 200\mu s$) in simulation improves goodput by an order of magnitude for both Reno and NewReno. *rto* represents runs with a modified RTO_{min} value.

5.1.3 Disabling TCP Slow-Start

Finally, we disable TCP *slow-start* to prevent network congestion produced by flows that exponentially increase their window sizes to discover link capacity following a timeout (or at the beginning of a TCP transfer). Figure 12 shows that forcing TCP flows to discover link capacity using only additive increase does not alleviate the situation. We leave an analysis of even more conservative congestion control algorithms for future work.

5.2 Reducing the Penalty of Timeouts

Because many of the TCP timeouts seem unavoidable (e.g. *Full Window Loss*, *Lost Retransmit*), here we examine reducing the time spent waiting for a timeout. While this approach can significantly improve goodput, this solution should be viewed with caution because it also increases the risk of premature timeouts, particularly in the wide-area [4]. We discuss the consequences of this effect below.

The penalty of a timeout, or the amount of time a flow waits before retransmitting a lost packet without the “fast retransmit” mechanism provided by three duplicate ACKs, is the retransmission timeout (RTO). Estimating the right RTO value is important for achieving a timely response to packet losses while avoiding premature timeouts. A premature timeout has two negative effects: 1) it leads to a spurious retransmission; and 2) with every timeout, TCP re-enters *slow-start* even though no packets were lost. Since there is no congestion, TCP thus would underestimate the link capacity and throughput would suffer. TCP has a conservative minimum RTO (RTO_{min}) value to guard against spurious retransmissions [29, 19].

Popular TCP implementations use an RTO_{min} value of 200ms [35]. Unfortunately, this value is orders of magnitude greater than the round-trip times in *SAN* settings,

which are typically around $100\mu s$ for existing 1Gbps Ethernet SANs, and $10\mu s$ for Infiniband and 10Gbps Ethernet. This large RTO_{min} imposes a huge throughput penalty because the transfer time for each data block is significantly smaller than RTO_{min} .

Figure 13 shows that reducing RTO_{min} from 200ms to $200\mu s$ improves goodput by an order of magnitude for between 8 to 32 servers. In general, for any given SRU size, reducing RTO_{min} to $200\mu s$ results in an order of magnitude improvement in goodput using TCP Reno (Figure 13(a)). Figure 13(b) shows that even with an aggressive RTO_{min} value of $200\mu s$, TCP NewReno still observes a 30% decrease in goodput for 64 servers.

Unfortunately, setting RTO_{min} to such a small value poses significant implementation challenges and raises questions of safety and generality.

Implementation Problems: Reducing RTO_{min} to $200\mu s$ requires a TCP clock granularity of $100\mu s$, according to the standard RTO estimation algorithm [29, 19]. BSD TCP and Linux TCP implementations are currently unable to provide this fine-grained timer. BSD implementations expect the OS to provide two coarse-grained “heartbeat” software interrupts every 200ms and 500ms, which are used to handle internal per-connection timers [5]; Linux TCP uses a TCP clock granularity of 1 to 10ms. A TCP timer in microseconds needs either hardware support that does not exist or efficient software timers [6] that are not available on most operating systems.

Safety and Generality: Even if sufficiently fine-grained TCP timers were supported, reducing the RTO_{min} value might be harmful, especially in situations where the servers communicate with clients in the wide-area. Allman et. al. [4] note that RTO_{min} can be used for trading “timely response with premature timeouts” but there is no optimal balance between the two in current TCP imple-

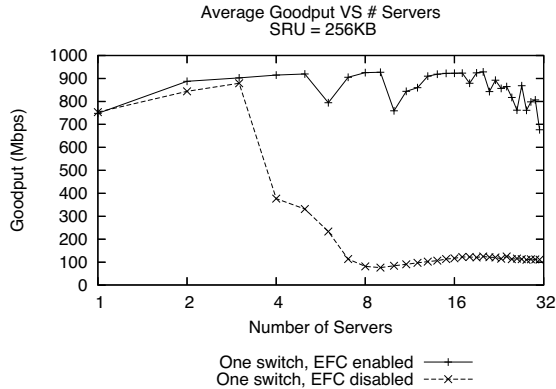


Figure 14: Enabling Ethernet Flow Control can mitigate *Incast* for a single-switch network.

mentations: a very low RTO_{min} value increases premature timeouts. Earlier studies of RTO estimation in similar high-bandwidth, low-latency ATM networks also show that very low RTO_{min} values result in spurious retransmissions [34] because variation in the round-trip-times in the wide-area clash with the standard RTO estimator's short RTT memory.

6 Ethernet Flow Control

Some Ethernet switches provide a per-hop mechanism for flow control that operates independently of TCP's flow control algorithm. When a switch that supports Ethernet Flow Control (EFC) is overloaded with data, it may send a "pause" frame to the interface sending data to the congested buffer, informing all devices connected to that interface to stop sending or forwarding data for a designated period of time. During this period, the overloaded switch can reduce the pressure on its queues.

We find that EFC is effective in the simplest configuration (i.e. all clients and servers connected to one switch), but does not work well with more than one switch, has adverse effects on other flows in all configurations, and is inconsistently implemented across different switches.

We measure the effect of enabling Ethernet Flow Control on a single HP Procurve 2848 switch, where one client and multiple servers are directly connected to the switch. Figure 14 shows that EFC can significantly improve performance. Unfortunately, TCP goodput is still highly variable and is lower than it would be without *Incast*.

Despite its potential benefits, our simple network topology and workload hide adverse side effects that surface when EFC is used on larger multi-switch networks with many more clients and active TCP flows. For many of these reasons, most switch vendors and network operators

keep EFC inactive.

The most significant problem with EFC is head-of-line blocking, which occurs when a pause frame originating from one congested interface stops several other flows from communicating simultaneously. The effects of head-of-line blocking can be particularly severe in heterogeneous bandwidth settings where one slow link can cause other faster links to be underutilized. In other words, pause frames pause *all* traffic entering an interface, regardless of whether that traffic is causing congestion.

Due to the complexities of head-of-line blocking, and because the particular interactions of EFC across multiple switches is inconsistently implemented across switch vendors, enabling EFC effectively across more than one switch can be a difficult or impossible task. For instance, in order to provide link aggregation between two HP Procurve 2848 switches, our system was configured with a virtual interface for the trunk – a configuration over which the switch did not support flow control.

While Ethernet Flow Control currently interacts adversely with other flows, a number of recent Ethernet initiatives have been introduced to add congestion management with rate-limiting behavior and to improve the pause functionality with a more granular per-channel capability [39]. These initiatives are part of a larger movement to create a lossless and flow-controlled version of Ethernet, referred to as *Data Center Ethernet*, which will allow the consolidation of multiple communication fabrics (including storage networks running Fibre Channel) into a single Ethernet solution.

Truly lossless behavior at the Ethernet level is a valid solution to the *Incast* problem, but it will take a number of years before these new standards are implemented in switches, and even then there are no guarantees that new switches will implement these standards uniformly or that they will be as commoditized and inexpensive as current Ethernet switches.

7 Related Work

Providing storage via a collection of storage servers networked using commodity TCP/IP/Ethernet components is an increasingly popular approach. The *Incast* problem studied comprehensively in this paper has been noted previously by several researchers (e.g., [15, 17, 25, 24]) while developing this cluster-based approach.

Nagle et al. briefly discussed the switch buffer overruns caused by clients reading striped data in a synchronized many-to-one traffic pattern [25]. Upgrading to better switches with larger buffer sizes was one adopted solution. They also mentioned the possibility of using link-level flow control, but highlight the difficulties of such an approach for different non-trivial switch topologies without

incorporating higher-level striping information used by the storage system.

In later work, Nagle et al. again report on the effects of *Incast* on scalable cluster-based file storage performance [24]. Specifically, they report on experiments with a production-quality system where a single client reads a file sequentially using an 8MB synchronization block size striped across multiple storage servers. As the number of storage servers is increased for their system, keeping all other variables of the network constant, the authors observe a linear scaling of storage bandwidth for up to 7 storage servers, a steady plateau until around 14 servers, and then a rapid drop-off. The primary cause of this performance collapse was attributed to multiple senders overwhelming the buffer size of the network switch. This prior work also observed that the *Incast* problem does not appear when running a streaming network benchmark like `netperf`. Therefore, the performance collapse is also attributed to the synchronized and coordinated reads in a cluster-based storage environment. Nagle et al. also discuss modest performance gains when using SACK or reducing the length of TCP retransmission timeouts. Although this last point is not quantified, they observe that degraded performance still persists even with these changes.

At a higher level, the *Incast* problem is a particular form of network congestion, a topic which has been studied extensively in different environments. Early work on congestion control in the wide-area network by Van Jacobson addressed the TCP congestion collapse of the Internet around 1985 [19]. Adopted as the basis of TCP congestion control, the idea was to provide a method for a networked connection to discover and dynamically adjust to the available end-to-end bandwidth when transferring data. Chiu and Jain describe why the window mechanism of “additive increase / multiplicative decrease” achieves fairness and stability in this setting [10].

Unfortunately, TCP’s congestion control and avoidance algorithms are not directly applicable to all settings. For example, they are known to have problems in wireless settings, where packet losses may not actually be caused by congestion. TCP also has problems in high-latency, high-bandwidth network settings [20]. The *Incast* problem provides another example of a network setting where using TCP may cause poor performance.

The performance and fairness of TCP when many flows share the same bottleneck was studied by Morris [23]. As the number of TCP flows through a bottleneck increases to the point where there are more flows than packets in the bandwidth-delay product, there is an increasingly high loss rate and variation of unfair bandwidth allocation across flows. This paper applies some of Morris’s methods and analysis techniques to the synchronized reads setting that produces *Incast*.

8 Conclusion

Incast occurs when a client simultaneously receives a short burst of data from multiple sources, overloading the switch buffers associated with its network link such that all original packets from some sources are dropped. When this occurs, the client receives no data packets from those sources and so sends no acknowledgement packets, requiring the sources to timeout and then retransmit. Often, the result of these TCP timeouts is an order of magnitude decrease in goodput.

Unfortunately, this traffic pattern is very common for the growing class of cluster-based storage systems. When data is striped across multiple storage nodes, each client read creates this pattern and large sequential reads create it repeatedly (once for each full stripe).

Whether or not *Incast* will cause goodput collapse in a system depends on details of the TCP implementation, network switch (especially buffer sizes), and system configuration (e.g., the number of servers over which data is striped). Unfortunately, avoiding collapse often requires limiting striping to a small number of servers. Techniques such as very short timeouts and link-level flow control can mitigate the effects of *Incast* in some circumstances, but have their own drawbacks. No existing solution is entirely satisfactory, and additional research is needed to find new solutions by building on the understanding provided by this paper.

Acknowledgments

We are grateful to Jeff Butler, Abbie Matthews, and Brian Mueller at Panasas Inc. for helping us conduct experiments on their systems. We thank Michael Stroucken for his help managing the PDL cluster. We thank our paper shepherd Ric Wheeler, Michael Abd-El-Malek, and all of our reviewers for their feedback. We also thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, Network Appliance, Oracle, Seagate, and Symantec) for their interest, insights, feedback, and support.

This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0546551, #CNS-0326453 and #CCF-0621499, by the Army Research Office under agreement number DAAD19-02-1-0389, by the Department of Energy under award number DE-FC02-06ER25767, and by DARPA under grant #HR00110710025. Elie Krevat is supported in part by an NDSEG Fellowship from the Department of Defense.

References

- [1] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa Minor: Versatile Cluster-based Storage. In *Proc. 4th USENIX Conference on File and Storage Technologies* (San Francisco, CA, Dec. 2005).
- [2] ALLMAN, M., BALAKRISHNAN, H., AND FLOYD, S. *Enhancing TCP's Loss Recovery Using Limited Transmit*. Internet Engineering Task Force, Jan. 2001. RFC 3042.
- [3] ALLMAN, M., AND PAXSON, V. *TCP Congestion Control*. Internet Engineering Task Force, Apr. 1999. RFC 2581.
- [4] ALLMAN, M., AND PAXSON, V. On Estimating End-to-End Network Path Properties. *SIGCOMM Comput. Commun. Rev.* 31, 2 supplement (2001).
- [5] ARON, M., AND DRUSCHEL, P. TCP Implementation Enhancements for Improving Webserver Performance. Tech. Rep. TR99-335, Rice University, June 1999.
- [6] ARON, M., AND DRUSCHEL, P. Soft timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems* 18, 3 (2000), 197–228.
- [7] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. Fail-Stutter Fault Tolerance. In *Proc. HotOS VIII* (Schloss-Elmau, Germany, May 2001).
- [8] BRAAM, P. J. File Systems for Clusters from a Protocol Perspective. <http://www.lustre.org>.
- [9] BUTLER, J. Personal communication, Mar. 2007.
- [10] CHIU, D.-M., AND JAIN, R. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems* 17 (1989), 1–14.
- [11] COMER, D. E. *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*. Prentice Hall, Englewood Cliffs, N.J., 2000.
- [12] FALL, K., AND FLOYD, S. Simulation-based Comparisons of Tahoe, Reno, and Sack TCP. *ACM Computer Communications Review* 26, 3 (July 1996), 5–21.
- [13] FLOYD, S., HENDERSON, T., AND GURTOV, A. *The NewReno Modification to TCP's Fast Recovery Algorithm*. Internet Engineering Task Force, Apr. 2004. RFC 3782.
- [14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)* (Lake George, NY, Oct. 2003).
- [15] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. 8th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, Oct. 1998).
- [16] GRIDER, G., CHEN, H., JUNEZ., J., POOLE, S., WACHA, R., FIELDS, P., MARTINEZ, R., KHALSA, S., MATTHEWS, A., AND GIBSON, G. PaScal - A New Parallel and Scalable Server IO Networking Infrastructure for Supporting Global Storage/File Systems in Large-size Linux Clusters. In *Proceedings of the 25th IEEE International Performance Computing and Communications Conference, Phoenix, AZ* (Apr. 2006).
- [17] HASKIN, R. High performance NFS. Panel: High Performance NFS: Facts & Fictions, SC'06.
- [18] HILDEBRAND, D., HONEYMAN, P., AND ADAMSON, W. A. pNFS and Linux: Working Towards a Heterogeneous Future. In *8th LCI International Conference on High-Performance Cluster Computing* (Lake Tahoe, CA, May 2007).
- [19] JACOBSON, V. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM* (Vancouver, British Columbia, Canada, Sept. 1998), pp. 314–329.
- [20] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion Control for High Bandwidth-Delay Product Networks. In *Proc. ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002).
- [21] LEISERSON, C. E. Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Transactions on Computers* 34 (Oct. 1985), 892–901.
- [22] MATHIS, M., MAHDAVI, J., FLOYD, S., AND ROMANOW, A. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force, 1996. RFC 2018.
- [23] MORRIS, R. TCP Behavior with Many Flows. In *IEEE International Conference on Network Protocols (ICNP)* (Oct. 1997).
- [24] NAGLE, D., SERENYI, D., AND MATTHEWS, A. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2004).
- [25] NAGLE, D. F., GANGER, G. R., BUTLER, J., GOODSON, G., AND SABOL, C. Network Support

- for Network-attached Storage. In *Hot Interconnects* (Stanford, CA, 1999).
- [26] NOUREDDINE, W., AND TOBAGI, F. The transmission control protocol: an introduction to tcp and a research survey. Tech. rep., Stanford University, 2002.
 - [27] ns-2 Network Simulator.
http://www.isi.edu/nsnam/ns/, 2000.
 - [28] PAWLOWSKI, B., AND SHEPLER, S. Network File System Version 4 (nfsv4) charter page.
 - [29] PAXSON, V., AND ALLMAN, M. *Computing TCP's Retransmission Timer*. Internet Engineering Task Force, Nov. 2000. RFC 2988.
 - [30] PETERSON, L. L., AND DAVIE, B. S. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 2003.
 - [31] POSTEL, J. B. *User Datagram Protocol*. Internet Engineering Task Force, Aug. 1980. RFC 768.
 - [32] POSTEL, J. B. *Internet Protocol*. Internet Engineering Task Force, Information Sciences Institute, Marina del Rey, CA, Sept. 1981. RFC 791.
 - [33] POSTEL, J. B. *Transmission Control Protocol*. Internet Engineering Task Force, Sept. 1981. RFC 793.
 - [34] ROMANOW, A., AND FLOYD, S. Dynamics of TCP traffic over ATM networks. *ACM Computer Communications Review* 24, 4 (1994), 79–88.
 - [35] SAROLAHTI, P., AND KUZNETSOV, A. Congestion control in Linux TCP. In *Proc. USENIX Annual Technical Conference* (Berkeley, CA, June 2002).
 - [36] SCHMUCK, F., AND HASKIN, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. USENIX Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002).
 - [37] SHEPLER, S., EISLER, M., AND NOVECK, D. NFSv4 Minor Version 1 – Draft Standard.
 - [38] STEVENS, W. R. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, 1994.
 - [39] WADEKAR, M. Enhanced Ethernet for Data Center: Reliable, Channelized and Robust. In *15th IEEE Workshop on Local and Metropolitan Area Networks* (June 2007).

Appendix – TCP Primer

We provide the reader with a brief background on TCP for the purposes of understanding the terms used in this paper. While we skip many of the detailed nuances of TCP, we refer the reader to several well known resources for further TCP details [33, 3, 11, 38, 30, 26].

The Transmission Control Protocol (TCP) [33] is a connection-oriented protocol that guarantees a reliable,

in-order byte-stream communication service between two processes, in contrast to the best-effort connectionless datagram delivery service provided by the User Datagram Protocol (UDP) [31]. Both TCP and UDP use the Internet Protocol (IP) [32], a best-effort datagram service, to carry their messages. The use of TCP is attractive for applications that perform pairwise communication as it offers the following advantages:

- Reliability – dealing with message loss, duplication, damage, and delay
- The in-order delivery of data
- Flow control and congestion control
- Multiplexing and demultiplexing to support multiple end-points on the same host through the use of port numbers

A TCP connection is *Full Duplex* – once a TCP connection is established between two end-points using a 3-way handshake protocol, the connection supports a pair of byte streams, one in each direction. TCP transfers a byte-stream by bundling together contiguous bytes into a TCP segment or packet.

A TCP packet, encapsulated in an IP packet, may be dropped en-route to the destination due to several causes, such as 1) the sender's kernel buffer being full, 2) a router buffer on the path to the destination being full, 3) routing errors, or 4) the receiver's kernel buffer being full. TCP uses a positive acknowledgement scheme with retransmissions to achieve reliable data transfer. To assist in the in-order delivery of data at the receiver, the TCP sender assigns a sequence number to every byte of data sent over a TCP connection. For a given packet, the sequence number assigned to the packet is the sequence number of the first byte within the packet. TCP uses a *cumulative* acknowledgment scheme: the ACK packet contains a sequence number informing the sender that it has received all bytes up to, but not including, that sequence number. While TCP assigns sequence numbers based on bytes, for simplicity, we discuss sequence numbers based on packets.

To make efficient use of the link, TCP uses a sliding window algorithm to keep multiple packets in flight. A window defines the number of packets that are unacknowledged: the left-edge of the window indicates the first packet not acknowledged by the receiver. For example, as shown in Figure 15, if the sender's window has a left-edge sequence number of 10 and a window size of 6, then the receiver has acknowledged the receipt of all packets with a sequence number less than 10, and the sender can transmit packets 10-15 all at once. When packets 10 and 11 reach the receiver, the receiver sends an ACK for packets 11 and 12 respectively. Since the left edge of the window now starts at 12 and the window size is 6, the sender may now transmit packets 16 and 17.

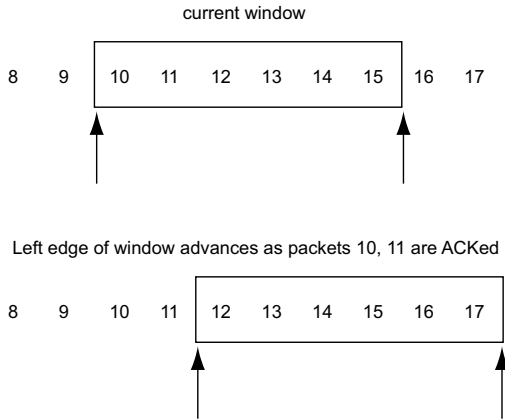


Figure 15: An illustration of TCP's sliding window mechanism with a fixed window size.

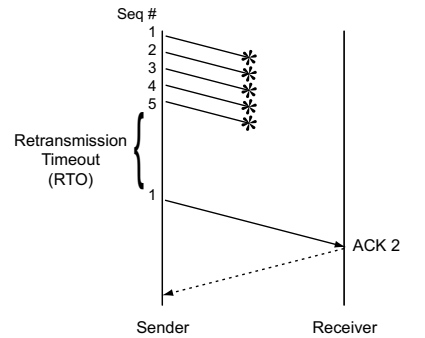
If a packet is delivered to a TCP receiver out-of-order, either due to re-ordering or losses in the network, the receiver generates a duplicate ACK for the last packet it received in-order. Building on the example above, if packet 12 was dropped by the network, then on receiving packet 13, the TCP receiver generates an ACK for packet 12 instead of packet 14, and the left edge of the window is not advanced.

A TCP sender detects a packet loss using the two schemes described below. Once a packet loss is detected, a TCP sender recovers from the loss by retransmitting the packet.

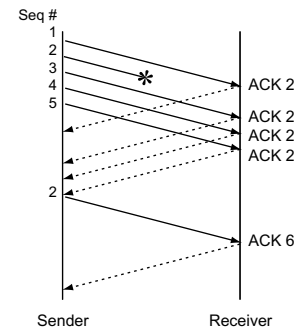
Timeout driven loss recovery: Consider the case shown in Figure 16(a). Packets 1 to 5 sent by the sender are all dropped. The sender waits for a certain amount of time, defined by the retransmission timeout (RTO), before a timeout event indicates the possible loss of packet 1, at which point the sender recovers from the loss by retransmitting packet 1. The RTO value is based on the round trip time (RTT), which is an estimated value.

Data driven loss recovery: A duplicate ACK can be used as an indication of loss, but it could also be generated due to packet reordering in the network. To distinguish benign reordering from actual loss, TCP senders normally consider 3 duplicate ACKs for a packet as an indication of a loss. Figure 16(b) shows a case where a sender transmits 5 packets but the second packet is lost. Packets 3, 4, and 5 generate duplicate ACKs indicating that packet 2 is lost. On getting 3 duplicate ACKs for packet 2, the sender assumes that packet 2 is lost and retransmits it: this is called *fast retransmit*. On receiving packet 2, the receiver ACKs packet 6, the next in-order packet it expects from the sender. Data-driven loss recovery responds to losses more quickly than timeout-driven recovery.

TCP provides end-to-end *flow control* whereby the receiver can control the amount of data a sender transmits.



(a) Timeout-driven Recovery



(b) Data-driven Recovery.

Figure 16: Recovery schemes in TCP

With every ACK, the receiver returns a window size indicating the number of packets a sender may transmit.

TCP is adaptive – flows utilize available bandwidth by probing the network. On startup and following a timeout, TCP has no good estimate of the capacity of the end-to-end path, so it enters *slow-start* to discover the capacity. For every ACKed packet received, the sender grows its window by 1. This results in an exponential growth in the window size and, hence, in the sending rate.

Congestion occurs when the sending rate exceeds the available bandwidth and packets are dropped. Various TCP algorithms have been designed to deal with congestion – they do this by reacting to congestion (indicated by loss) by throttling the rate at which the sender transmits data. Under data-driven loss recovery, the sender performs a *multiplicative decrease* by halving its window (also accompanied by *fast-recovery*) and begins an *additive increase* (or congestion avoidance) phase, where for every window of data acknowledged by the receiver, the sender increases its window size by 1. Under timeout-driven recovery, the sender reduces its window to 1, performs slow-start until a certain threshold, and then enters the congestion avoidance phase.

Portably Solving File TOCTTOU Races with Hardness Amplification

Dan Tsafir
IBM Research
Yorktown Heights, NY
dants@us.ibm.com

Tomer Hertz
Microsoft Research
Redmond, WA
hertz@microsoft.com

David Wagner
UC Berkeley
Berkeley, CA
daw@cs.berkeley.edu

Dilma Da Silva
IBM Research
Yorktown Heights, NY
dilmasilva@us.ibm.com

Abstract

The file-system API of contemporary systems makes programs vulnerable to TOCTTOU (time of check to time of use) race conditions. Existing solutions either help users to detect these problems (by pinpointing their locations in the code), or prevent the problem altogether (by modifying the kernel or its API). The latter alternative is not prevalent, and the former is just the first step: programmers must still address TOCTTOU flaws within the limits of the existing API with which several important tasks can not be accomplished in a portable straightforward manner. Recently, Dean and Hu addressed this problem and suggested a probabilistic hardness amplification approach that alleviated the matter. Alas, shortly after, Borisov et al. responded with an attack termed “filesystem maze” that defeated the new approach.

We begin by noting that mazes constitute a generic way to deterministically win many TOCTTOU races (gone are the days when the probability was small). In the face of this threat, we (1) develop a new user-level defense that can withstand mazes, and (2) show that our method is undefeated even by much stronger hypothetical attacks that provide the adversary program with ideal conditions to win the race (enjoying complete and instantaneous knowledge about the defending program’s actions and being able to perfectly synchronize accordingly). The fact that our approach is immune to these unrealistic attacks suggests it can be used as a simple and portable solution to a large class of TOCTTOU vulnerabilities, without requiring modifications to the underlying operating system.

1 Introduction

The TOCTTOU (time of check to time of use) race condition was characterized in 1974 by McPhee as the situation which occurs

“if there exists a time interval between a validity-check and the operation connected with that validity-check [such that], through multitasking, the validity-check variables can deliberately be changed during this time interval, resulting in an invalid operation being performed by the control program.” [25]

Dissecting a 1993 CERT advisory [7], Bishop was the first to systematically show that file-systems with weak consistency semantics (like Unix and Windows) are inherently vulnerable to TOCTTOU races [3, 4]: First, a program checks the status of a file using the file’s name. Then, depending on the status, it applies some operation to the file, unjustifiably assuming the status has not changed since it was checked. This error is caused by the fact that the mapping between file names and file objects (“inodes”) is mutable by design, and might therefore change between a status check and a subsequent operation.

Researchers have put a lot of effort into trying to solve or alleviate the problem, (1) developing compile-time tools to pinpoint locations in the source code that are suspect of suffering from a TOCTTOU race [4, 37, 10, 8, 30], (2) modifying the kernel to log all relevant system calls and analyzing the log, postmortem, to detect TOCTTOU attacks [20, 16, 21, 19, 39, 1], (3) having the kernel speculatively identify offending processes and temporarily suspend them or fail their respective suspected system calls [11, 34, 27, 35, 28], and finally (4) designing new file-system interfaces to make it easier for programmers to avoid the races. [3, 29, 24, 40].

None of the above helps programmers to safely and portably accomplish a TOCTTOU-prone task on *existing* systems, as kernels that prevent races are currently an academic exercise, whereas new-and-improved file-systems are unfortunately not prevalent (and certainly not standard). Thus, regardless of how programmers become aware of the problem, whether through compile-time tools or just by being careful, they must still face the problem with the existing API.

At the same time, resolving a TOCTTOU race is not as easy as, e.g., fixing a buffer overflow bug, because the

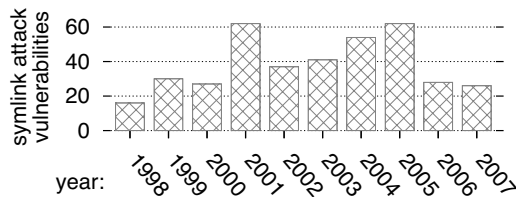


Figure 1: NVD reports 450 “symlink attack” vulnerabilities, as of September 5, 2007. (In 2001 and 2005 there were 73 and 106 reports, respectively; the associated bars are truncated.)

programmer must somehow achieve atomicity of two operations using an API that was not designed for such a purpose. In fact, overcoming TOCTTOU races in a portable manner is notoriously hard, sometimes even for experts (see Section 2.3). Hence, it is probably impractical to expect average programmers to successfully accomplish such tasks (or attempt them) on a regular basis.

Indeed, to date, TOCTTOU races pose a significant problem, as exemplified by Wei and Pu, which analyzed CERT [36] advisories between 2000 and 2004 and found 20 reports concerning the issue, 11 of which provided the attacker with unauthorized root access [39]. Figure 1 shows the yearly number of TOCTTOU “symlink attack” vulnerabilities reported by NVD (National Vulnerability Database) [26]. These affect a wide range of mainstream applications and tools (e.g., bzip2, gzip, FireFox, make, OpenOffice, OpenSSL, Kerberos, perl, samba, sh), environments (e.g., GNOME, KDE), distributions (e.g., Debian, Mandrake, RedHat, SuSE, Ubuntu), and operating systems (e.g., AIX, FreeBSD, HP-UX, Linux, Solaris).

We contend that the situation can potentially be greatly improved if programmers are able to use some portable, standard, generic, user-mode `check_use` utility function that, given a ‘check’ operation and a ‘use’ operation, would perform the two as a kind of “transaction”, in a way that appears atomic for all relevant purposes. This paper takes a significant step towards achieving such a goal.

The first step in this direction was taken in 2004 by Dean and Hu, which implemented a transaction-like `access_open` routine that set out to solve a single race [12]: the one which occurs between the `access` system call (used by root to check if a user has adequate privileges to open a file) and the subsequent `open`. Their idea (later termed *K-race* [5]) was to use *hardness amplification* as found in the cryptology literature [41], but applied to system calls rather than cryptologic primitives. In a nutshell, if an adversary has a probability $p < 1$ to win a race, then the probability p^K to win K races can be made negligible by choosing a big enough K . Indeed, by mandating attackers to win K consecutive races before agreeing to `open` the file, `access_open` seemingly accomplished its “transactional” goal of aggregating `access` and `open` into

a single “atomic” operation.

But the new and intriguing *K-race* defense did not stand the test of time. In 2005, Borisov et al. orchestrated their *filesystem maze* attack and showed that an adversary can in fact win *every* race (hence making the assumption that $p < 1$ wrong) [5]. Roughly speaking, the adversary is able to slow down, and effectively “single step”, the proposed algorithm by feeding it with a carefully constructed file name (the “maze”) and polling the status of certain components within the name. This induces perfect synchronicity between the adversary and the *K-race*, thereby enabling the adversary to win all races ($p \approx 1$). Indeed, in his on-line publication list, adjacent to his 2004 paper [12], Alan Hu concedes that

“The scheme proposed here has been beautifully and thoroughly demolished by Borisov, Johnson, Sastry, and Wagner [5]. The theory is, of course, still valid, but it relies on an assumption of the attacker having a non-negligible probability of losing races. Borisov et al. came up with ingenious means (1) to force the victim to go to disk on each race, thereby allowing plenty of time for the attacker to win races, and (2) to determine precisely what protocol operation the victim is doing at any point in time, thereby foiling the randomized delays. The upshot is that they can win these TOCTTOU races with almost complete certainty.” [17]

Dean and Hu were only concerned with finding a way to correctly use the `access` system call; likewise, the explicit goal of Borisov et al. was to prove that `access` should never be used. But the consequences of the filesystem maze attack are much more general. In fact, mazes constitute a generic way to consistently win a large class of TOCTTOU races. This is true because any ‘check’ operation can be slowed down and single-stepped, if provided with a filesystem maze as an argument. Consequently, the common belief that “TOCTTOU vulnerabilities are hard to exploit, because they [...] rely on whether the attacking code is executed within the usually narrow window of vulnerability (on the order of milliseconds)” [39] is no longer true: With filesystem mazes, the attacker can often proactively prolong the vulnerability window, while simultaneously finding out when it opens up.

Motivated by the alarmingly wide applicability of the filesystem maze attack, we set out to search for an effective defense, with the long-term goal of providing programmers with a generic and portable `check_use` utility function that would allow for a pseudo-atomic transaction of the ‘check’ and ‘use’ operations. Importantly this should work on existing systems, without requiring changes to the kernel or the API it provides.

This paper is structured as follows: After exemplifying the TOCTTOU problem in detail, surveying the existing

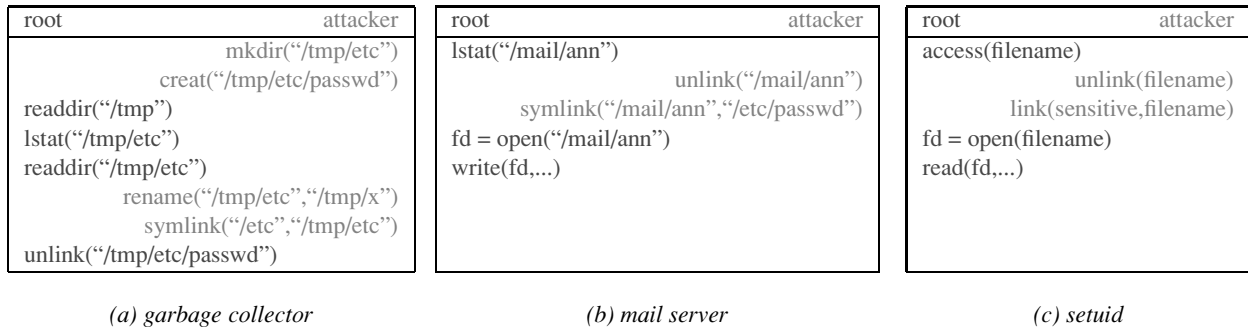


Figure 2: Three canonical file TOCTTOU examples. The Y-axis denotes the time (future is downwards). The left-justified operations, performed by root, suffer from a TOCTTOU vulnerability. The right-justified operations show how an attacker can exploit this vulnerability to circumvent the system’s protection mechanisms and to gain illegal access.

solutions, and pointing out their shortcomings and the elusiveness of a contemporary practical solution (Section 2), we go on to explain how hardness amplification was applied to solve file TOCTTOU races, and why it has failed (Section 3). We then show how to turn this failure to success (Section 4) and experimentally evaluate our solution by subjecting it to a hypothetical attack far more powerful than filesystem mazes (Sections 5–6). We discuss how to generalize our solution, its limitations, and how/when its probabilistic aspect can be eliminated (Section 7). Finally, we present our conclusions (Section 8).

2 Motivation

Much of the administrative and security-crucial tasks of Unix-like systems is performed by root-privileged programs. Since such programs often interact with and affect the system by means of file manipulation, they are susceptible to TOCTTOU vulnerabilities. A successful exploitation of these vulnerabilities would allow a non-privileged user to circumvent the system’s normal protection mechanisms and unlawfully execute some operation as root.

2.1 Classic Examples

For example, many sites periodically delete files residing under the `/tmp` directory. If a file was not accessed for a certain amount of time, the “garbage collection” script deletes it. Mazières and Kaashoek noted that this policy might contain a TOCTTOU window between the ‘check’ statement (of the file access time) and the subsequent ‘use’ statement (the file removal); if a name/inode mapping changes within this window, the script can be tricked into deleting any arbitrary file, even if it attempts to prevent this from happening by explicitly ignoring symbolic links [24]. This is illustrated in Figure 2a: The garbage collector uses `lstat` to verify that `/tmp/etc` is not a symbolic

link. But as with all TOCTTOU flaws, this check is fruitless in case `/tmp/etc` is manipulated just after.

Another well known TOCTTOU example, initially documented by Bishop, is that of a mail server which appends a new message to the corresponding user’s Inbox file [3, 4]. Before `open`-ing the Inbox, the server `lstat`-s it to rule out the possibility the user has replaced it with some symbolic link pointing to a file that lies elsewhere. Figure 2b shows how the inevitable associated TOCTTOU race can be exploited to add arbitrary data to the `/etc/passwd` file, providing the attacker with the ability to obtain permanent root access.

A third example concerns the *setuid bit* that Unix-like systems associate with an executable to indicate it should run with the privileges of its *owner*, rather than the user that *invoked* it (as is the normal case). Of course just handing off root privileges is not a good idea, which is why the `access` system call conveys `setuid` programs the ability to check whether an invoker has adequate privileges:

```
if ( access(filename, R_OK) == 0 )
    fd = open(filename, O_RDONLY);
```

Alas, the `access/open` idiom constitutes the archetypal, and arguably the most infamous, TOCTTOU flaw.¹ Figure 2c illustrates how this race can be exploited to access any file; `access` was therefore deemed unusable, as e.g. indicated by its FreeBSD manual, explicitly stating that “the `access` system call is a potential security hole due to race conditions and *should never be used*.” [22]

2.2 Existing Solutions

Considerable research effort have been put into providing solutions for TOCTTOU vulnerabilities like the ones described above. In order to highlight the contribution of

¹This race was reported by what is believed to be the first formal documentation of a file TOCTTOU vulnerability [7]; it is described by almost all papers that address the TOCTTOU issue (see Section 2.2) when exemplifying the problem.

this paper we first survey this work, which can be subdivided into four categories:

Static Detection Some groundbreaking work has been done in recent years to statically analyze the source code of programs and pinpoint the locations of nontrivial vulnerabilities and bugs [14, 15, 2, 13]. This type of analysis is rooted in Bishop’s work, which used pattern matching to locate pairs of TOCTTOU system calls in root-privileged programs on a per-function basis [3, 4]. The tools ITS4 [37], Eau Claire [10], and MOPS [8, 30] have later superseded Bishop’s work by being more general, accurate, and scalable.

Dynamic Detection Static analysis can be very effective and has the advantage of (1) not incurring runtime overheads, (2) covering all the code (in a reasonable amount of time), and (3) locating the bugs before the system is deployed. But the code is not always available, and even if it is, the static doctrine is inherently missing key information that is often only available at runtime, which might result in many false positives. To solve this, Ko and Redmond patched the kernel to log the required information and utilized it, postmortem, to feed a model that detects TOCTTOU flaws [20]. A similar approach was later adopted by many following projects [16, 21, 19, 39, 1]. Notable of these is the work by Wei and Pu [39] that exhaustively enumerated all of Linux’s TOCTTOU pairs² and the revolutionary IntroVirt tool by Joshi et al. [19] that made ubiquitous virtual-machine checkpointing and replaying a realistic alternative that can e.g. be used to identify TOCTTOU attacks, postmortem.

Dynamic Prevention The kernel can be modified to apply the principles of dynamic detection on-the-fly, as discovering TOCTTOU attacks while they occur allows for on-line prevention. This approach was first taken by Cowan et al. in 2001, when implementing “RaceGuard” [11]. Their technique tackles one TOCTTOU flaw that occurs between (1) a check if a candidate name for a temporary file doesn’t match an exist file, and (2) the new file’s creation (stat/open). They modify the kernel to maintain a cache of files that have been `stat`d and found not to exist; if a subsequent `open` finds an existing file, it fails.

In 2003, Tsyrlkevich and Yee developed a more general approach that was capable of generically preventing most TOCTTOU attacks [34]. They patched the kernel to

suspend any process that interferes with a “pseudo transaction” (check/use pair that agree on the target file), such that the worst outcome of a false-positive detection is a temporary suspension of the corresponding process. Several similar solutions followed [27, 35, 28]; the latter of which, by Pu and Wei, was argued to be “complete”, being based on their aforementioned earlier work [39].

New API All of the above are solutions that respect the existing file-system API so as to accommodate existing applications and operating systems. The complementary approach is to augment or change the API, such that tasks that currently suffer from TOCTTOU issues are made easier to safely accomplish. For example, to resolve the access/open race, Dean and Hu suggested that `open` would accept an `O_RUID` flag, which would instruct it to use the real (rather than effective) user ID of the process [12]; alternatively, Bishop suggested to add a new `faccess` system call that would operate on a file-descriptor rather than a file name [3].³ Likewise, the `O_NOFOLLOW` flag supported by Linux and FreeBSD makes `open` fail if its argument refers to a symbolic link, which may help in certain cases (e.g. Figure 2b). However, aside from being non-portable, it relates only to the last component of the file path: earlier components may still be symbolic links, and hence be juggled by an attacker (e.g. Figure 2a).

To obtain a more general solution, a bigger change is needed, such as replacing (or augmenting) Unix semantics with that of a transactional file-system [29, 40]: Atomicity would then insure that a check/use pair that was annotated by the programmer as a single transaction would be executed with no interference.

A more radical approach was suggested by Mazières and Kaashoek [24]. They proposed to use the fact that the binding between file descriptors and inodes is immutable (and thus cannot be exploited) to devise a safer programming paradigm that would make it harder for the programmer to make mistakes. By this paradigm,

1. all access checks would be done on file descriptors rather than on names,
2. users would be given explicit control of whether symlinks are followed when files are opened, and
3. each system call invocation would be provided with the user credentials with which the system call should operate.

We contend that some of this vision can be realized in user-mode on current systems.

²Wei and Pu (and later Lhee and Chapin [21]) augmented the definition of check/use TOCTTOU pairs to also refer to use/use pairs. With this, they found a bug in `rpm` that (1) generated a script that was writable by all (first use of `open`), and (2) executed it with root privileges (second use of `open`). While such bugs can be very hard to detect, they are nevertheless very easy to fix and therefore are of no interest in this paper.

³We note in passing that even though this suggestion was raised again by Dean and Hu, we contend it is impossible: the corresponding inode can possibly be refereed to by multiple paths, among which some are accessible to the user and some are not.

2.3 The Problem

Notice that all the existing solutions surveyed above *do not help programmers in resolving a known TOCTTOU flaw within existing systems*. Static detection techniques are invaluable in locating such flaws, but what are programmers to do if/once they are aware of the vulnerability? Surely they cannot wait until all contemporary kernels employ dynamic prevention (if ever, as significant complexity and performance penalty might be involved). Likewise, programmers cannot wait until all contemporary OSs portably support transactional file-systems (or constructs like the aforementioned API suggested by Mazières and Kaashoek).

The fact of the matter is that, in order to achieve a portable solution, programmers are bound to handling the matter with a decades-old API. Importantly, as mentioned earlier, a portable user-mode solution to a given TOCTTOU race (if exists) is often much harder and more elusive than e.g. fixing a buffer overflow bug: even experts that explicitly target a specific TOCTTOU problem are prone to getting it wrong.

Consider for example the access/open race depicted in Figure 2c. Tsyklevich and Yee suggested two solutions to this flaw [34]. The first argues that “to avoid this race condition, an application should change its effective id [with `setuid` system calls] to that of a desired user and then make the `open` system call directly.” However, after carefully evaluating this suggestion, Dean and Hu found that

“Unfortunately, the `setuid` family of system calls is its own rats nest. On different Unix and Unix-like systems, system calls of the same name and arguments can have different semantics, including the possibility of silent failure [9]. Hence, a solution depending on user id juggling can be made to work, but is generally not portable.” [12]

The second suggestion by Tsyklevich and Yee was “to use `fstat` after the `open` instead of invoking `access`”. As the input of `fstat` is a file descriptor, the latter is permanently mapped to the underlying inode and hence can never be abused by an attacker; the user is then expected to inspect the ownership information returned by `fstat` and check if the invoker was indeed allowed to `open` the file. But this will not work, as file access permissions can *not* be deduced in such a way; rather, they are the conjunction of all the (inode) permissions associated with each component in the respective path. For example, if a file’s name is `x/y` such that `x` is solely accessible by its owner, then other users are forbidden from reading `y` even if `fstat` indicates it is readable by all (which may very well be the case when root invokes the `fstat`).

A third alternative is to `fork` a child that permanently drops all extra privileges and then attempts to `open` the

file; if successful, the child can then pass the open file descriptor across a Unix-domain socket and `exit`. Borisov et al. [5] have mistakenly attributed the claim that this version is portable, to Dean and Hu [12]. But the latter have actually argued the contrary, stating that, with respect to the Unix-domain approach, “some of the above [user id juggling] caveats still apply”. Indeed, as mentioned earlier, dropping privileges is a non-portable operation [9]. (Regardless of whether it is being done by a parent or a forked child.) Furthermore, we find that passing an open descriptor alone, even without dropping privileges, suffers from serious portability issues.⁴

A fourth failed attempt will be discussed next.

3 Failure of Hardness Amplification

In 2004, noting that no prior art helps programmers to portably resolve TOCTTOU vulnerabilities on existing systems, Dean and Hu took the first step towards a portable solution [12], explicitly focusing their efforts on the aforementioned access/open TOCTTOU race.

3.1 The *K*-Race Technique

Their solution, termed “*K*-race”, was inspired by the hardness amplification technique that is commonly used in cryptography contexts [41]. The idea underlying hardness amplification is to use a problem which is computationally “somewhat hard”, in order to devise another computational problem that is “really hard”. In a TOCTTOU access/open scenario, the “somewhat hard” problem is timing and completing the attack (removing one file and linking another) within the exact window of opportunity delimited by the `access` and `open` calls (see Figure 2c). The “really hard” problem is requiring the attacker to succeed in doing this for $2K + 1$ consecutive times.

The *K*-race routine, shown in Figure 3, starts with a standard call to `access`, followed by an `open`, followed by *K* *strengthening rounds*. Each round consists of an additional `access` check and a corresponding `open`, which are then followed by a statement that verifies that the currently `opened` file is the same file that was `opened` in the previous round. Note that when $K = 0$, the routine degenerates to the standard access/open TOCTTOU race.

⁴ This is the result of changes related to the `msg_hdr` structure, which is used by the `sendmsg` and `recvmsg` system calls to pass an open descriptor through a Unix domain socket. Specifically, (1) in the mid 1990s, POSIX replaced the `msg_accrights` field with the `msg_control` array (but commercial OSes such as Solaris and HP-UX preferred to keep the earlier version as the default) and (2) more recently, RFC 3542 defined a set of macros to be exclusively used when accessing / manipulating the `msg_control` array (but despite being mandated by OSes like Linux, some of the macros are not yet standard) [33]. The end result is lack of portability and source code that is littered with `ifdefs` and conditional compilation tricks [32, 42, 31, 6].

```

#define DO_SYS(call) if((call)==-1) return -1
#define DO_CHK(expr) if( !(expr) ) return -1
#define DO_CMP(x,y) \
    ( ((x)->st_ino == (y)->st_ino) && \
      ((x)->st_dev == (y)->st_dev) )

int access_open_2004(char *fname)
{
    int fd1, fd2, i;
    struct stat s1, s2;

    // 1- the access/open idiom
    DO_SYS( access(fname, R_OK) );
    DO_SYS( fd1 = open (fname, O_RDONLY) );
    DO_SYS( fstat (fd1, &s1) );

    // 2- the strengthening rounds
    for(i=0; i<K; i++) {
        DO_SYS( access(fname, R_OK) );
        DO_SYS( fd2 = open (fname, O_RDONLY) );
        DO_SYS( fstat (fd2, &s2) );
        DO_SYS( close (fd2) );
        DO_CHK( DO_CMP(&s1, &s2) );
    }

    return fd1;
}

```

Figure 3: The *K*-race routine employs hardness amplification to probabilistically solve a TOCTTOU race. Specifically, on each strengthening round, it checks that the caller still has appropriate access permissions and that the underlying file-object, as represented by the inode (*st_ino*) and IO device (*st_dev*), remains the same. This attempts to provide programmers with a way to invoke *access* and *open* in an “atomic” manner.

To be successful, an attacker must indeed win $2K + 1$ races: This is true because, on each round, the *access* check must be applied to some user accessible file, or else permission is denied; On the other hand, every *open* must be applied to the same inaccessible target file, or else the verification that all file-descriptors refer to the same file-object would fail. Thus, assuming each race is an independent random event with some probability $p < 1$ for the attacker to win, the overall probability of tricking a *K*-race is p^{2K+1} . (Independence of events is supposedly obtained by introducing short random delays between successive system call invocations: as delays are randomized, an adversary wouldn’t be able to synchronize with the *K*-race.) After measuring several systems (among which are SMP systems), Dean and Hu concluded that $K=7$ is enough to make the probability of success negligible for all practical purposes.

3.2 Filesystem Mazes

In 2005, Borisov et al. defeated the *K*-race technique [5]. They have done so by refuting the (then widely accepted) assumption that the probability p for an attacker to win a

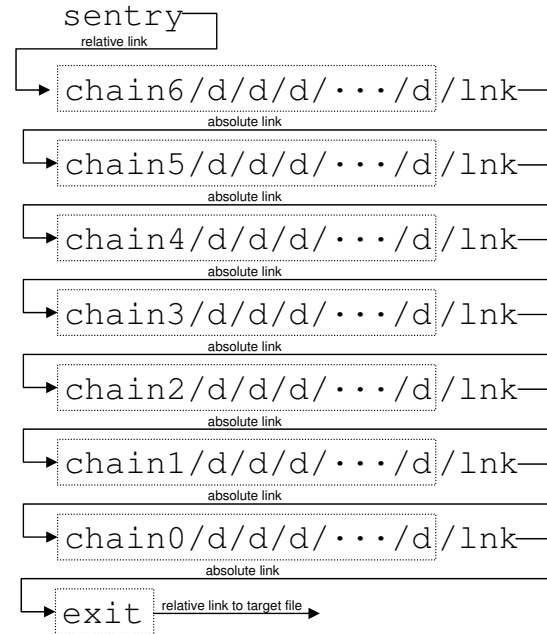


Figure 4: The structure of a six-chains filesystem maze. Arrows represents symbolic links. (Originally published in [5]; reprinted with permission.)

race is significantly smaller than one. In fact, they have managed to effectively make it a certainty ($p \approx 1$). The heart of the attack consists of a *filesystem maze*, which, in simple terms, is the longest and most nested filepath a user can pass as an argument to a system call, without causing it to fail due to hardcoded kernel limits.

Constructing a Maze The basic building block of a maze is a *chain*, defined to be (nearly) the deepest nested directory tree one can define without violating the *PATH_MAX* constraint imposed by the kernel on the length of file paths (4KB is a typical value). Thus, *chain₀* would be *chain0/d/d/d/.../d* such that the associated number of characters is a bit less than *PATH_MAX*. Likewise, *chain₁* is *chain1/d/d/d/.../d*, etc.

To form a maze, the attacker connects chains by placing a symbolic link at the bottom of *chain_{i+1}* that points to *chain_i*. The final symlink, at the bottom of *chain₀*, points to an *exit* symlink which, in turn, points to the actual target file. Finally, the entry point to the maze, *sentry*, is a symlink pointing to the highest chain. This is illustrated in Figure 4.

Unix systems impose a limit on the total number of symlinks that a single filename lookup can traverse, e.g., Linux 2.6 limits this number to 40. This places a limit on the number of chains composing the maze. Still, even with this limit, a maze can be composed of nearly 80,000

directories which may require loading about 300MB from the disk, just to resolve the associated name.

Importantly, if even one of the corresponding directory entries is not found in-memory, in the filesystem cache, the process that invoked the system call on behalf of which the path resolution is performed would be put to sleep, blocked-waiting for IO.

The Attack We now describe how to trick the K -race routine (Figure 3) into opening a private inaccessible file. The routine invokes `access` and `open` $K+1$ times. For these total of $2K+2$ invocations, we create $2K+2$ directories `dir1`, `dir2`, ..., `dir2K+2`, each containing a new maze. We arrange things such that `exit` points of odd mazes point to some public accessible file, whereas `exit` points of even mazes point to the inaccessible protected file we are about to attack. Finally, we generate a new symlink called `activedir` to point to `dir1`.

The attack is started by invoking the `access_open` K -race routine with the following filepath as an argument

```
activedir/sentry/lnk/lnk/.../lnk
```

This filepath is then passed along to the initial `access` call, which forces the K -race routine into the first maze. As a result, two things occur

1. The kernel updates the `atime` (access time) of every symbolic link it traverses during the name resolution, so by repeatedly examining the `atime` of `activedir/sentry` the attacker can learn that the respective `access` invocation is already in flight.
2. As mentioned earlier, the filepath being resolved (the maze) is big enough to insure that the kernel would have no choice but to fetch some of the relevant directory entries from disk; whenever this occurs the K -race routine would be suspended and put to sleep, and the attacker would get a chance to run and poll the `atime` of `activedir/sentry`.

Upon noticing that the `atime` has been updated, the attacker knows that the first `access` has begun. The attacker therefore switches `activedir` to point to `dir2`, and begins polling the `atime` of `dir2/sentry`. The initial `access` call is not affected by the change to `activedir` because it has already traversed that part of the path.

Eventually, the IO operations complete and the `access` finishes successfully. When the K -race calls the subsequent `open`, the exact same scenario occurs: the kernel updates the `atime` of `dir2/sentry`, the K -race routine sleeps on IO when loading parts of the respective maze that are not cached, the attacker consequently resumes and notices the updated `atime` of `dir2/sentry`, the attacker switches `activedir` to point to `dir3`, and the K -race routine completes the `open` successfully. This sequence

of events repeats itself until all the system calls composing the K -race complete, and the attacker has managed to fool the K -race and open the protected file.

Enhancements In order to increase the confidence that some directory entries are not cached by the filesystem while the name resolution takes place, an attacker can run in parallel various unrelated IO intensive activities to wipe out the cache. A recursive string search in the filesystem

```
grep -r anystring /usr > /dev/null 2>&1
```

was found to be especially effective in this respect.

Finally, for completeness, Borisov et al. considered a K -race version that randomly flips the order of the calls to `access` and `open` within the strengthening loop (this is a valid and technically sound defense against their maze attack). They defeated this approach as well, by deducing which system call is currently being executed with the help of various kernel variables exported through the `/proc` file-system. For example, in Solaris 9, any process can read the current system call number of any other process from `/proc/pid/psinfo`.

4 Making Amplification Work

The maze attack is a generic way to systematically win TOCTTOU races. By utilizing complex file names, an attacker can slowdown the victim application, effectively single-step it, and gain a decisive advantage, which allows it to defeat the probabilistic K -race approach. In this section we show that this advantage is in fact *not* inherent. Defenders need not play by the rules that are dictated by the attacker. Rather, they can impose new rules that make it practically impossible for an attacker to win.

The key observation is simple and well known: system calls like `open`, `stat`, `chdir`, `access`, `chown` etc. that operate on a specified file name, are in fact $O(n)$ algorithms, where n is the number of components composing the name (n also embodies symlinks that are part of the name as well as the components of the soft links that must be recursively traversed). And so, in order to resolve an n -component name, the associated system call must sequentially iterate through n inodes. In the case of the K -race approach this is done K times, so the number of traversed inodes is actually $n \cdot K$. The order in which the traversal is performed is crucial for the success of the maze attack; assuming a file name of the form `/f1/f2/f3` (with no symbolic links along the way) and assuming $K = 2$, this order would be:

`/, f1, f2, f3, /, f1, f2, f3`

The general case is illustrated in Figure 5 (left); due to this type of a visualization we call this order *row-oriented*.

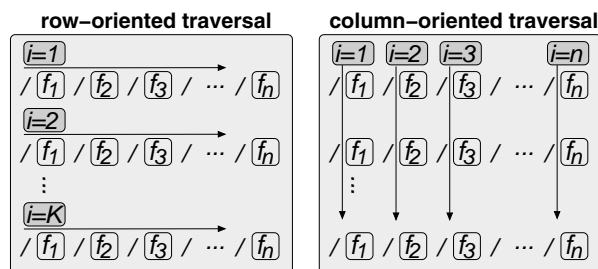


Figure 5: *The original row-oriented K -race traversal suggested by Dean and Hu (left) vs. our newly proposed column-oriented traversal (right). While Dean and Hu traverse the entire path on each access/open invocation, we traverse the path component by component, iterating through each specific element K times.*

The success of the K -race approach relies on the assumption that the rows remain identical from round to round. In contrast, the principle underlying the file-maze attack is to make n so big such that the time period between two “consecutive visits” in the inode associated with f_i would be relatively long; long enough to make it easy to violate the said assumption.

Our approach contends that row-orientated traversal, while seemingly dictated by the system call API, is not carved in stone. There is actually no technical difficulty preventing us from doing a different inode traversal that would better suit our needs. Specifically, column-oriented traversal is perfectly aligned with our intent to make it harder for an adversary to win a race. This approach is illustrated in Figure 5 (right). The idea is to resolve a path one component at a time, atom by atom, such that on each step we effectively conduct a kind of “short race” or “atom race”, as part of the K -strengthening doctrine. This approach provides a clear advantage: an adversary no longer has control over the duration of the elapsed time between consecutive visits at f_i , e.g. the traversal order in the above example would be:

/, /, f_1 , f_1 , f_2 , f_2 , f_3 , f_3

Thus, the race is made “fair” again and the respective inode would most probably be continuously present in the cache throughout the K -race, and almost certainly at least once during two consecutive iterations (which would be enough to defeat an attacker). The next section will show that even under the theoretical scenario where the attacker is *completely* and *instantaneously* synchronized with the defender, the attacker would have to wait tens to millions of years in order to subvert a $K = 9$ column-oriented defense.

We will now describe our algorithm in a bottom-up fashion (*all* source code included, as an indication of its simplicity). Doing a column-oriented traversal entails a price, which is having to handle the parsing of the file

path ourselves when splitting it into atoms. For our purposes, however, the `chop_1st` function (as listed in Figure 6) was all that was needed in this respect. This function gets a relative path and “chops off” the first component while returning the remainder to the caller. By repeatedly invoking this function (using the remainder of the path from the previous invocation as the input to the current invocation), we gradually consume the file path in a column-oriented manner.

A second difficulty one faces when doing a user-level path resolution is having to handle atom components that are in fact symbolic links. To handle this caveat we used the simple `is_symlink` function (listed in Figure 7) that gets as input the atom that was just chopped off the prefix of the full file path. Note that by applying the `lstat` system call upon the given atom we make sure that the invoker is not forced to go through a maze. If this atom happens to be a symbolic link, then `is_symlink` copies the name of the target file to the memory pointed to by the appropriate argument; this would be later processed recursively. However, if the atom is a hard link (read: not a symlink), then the result of the `lstat` operation (as recorded by the given `stat` structure) will be used as a reference point within the race, when inodes are compared, as described next.

Having dealt with all the low-level details, we go on to consider how a race would actually be conducted when a hard link is finally encountered. Recall that the access permissions of a file are more than just the per-inode access bits (user/group/all read/write/execute etc.): they are the conjunction of all the permissions of each and every directory component along the path. For example, even if an inode indicates it is readable by all, if it nevertheless resides within a private directory, then obviously no one should be able to access the associated file. Therefore, before descending into the next directory component, the algorithm must verify that the invoker has the appropriate permissions. However, since this entails a TOCTTOU vulnerability, each such check must be K -strengthened.

Figure 8 shows how a per-atom K -race is conducted. Note that the security of our algorithm is reduced to the security of `atom_race` (all other functions are completely safe). The information encapsulated by the `stat` structure input was placed there by the `is_symlink` function that has just been invoked using the very same atom. Thus, it is likely that the inode (that is associated with the atom) is still in the cache. Further, since the atom is in fact an “atom” (one component file) that has just now been verified to be a hard link, it is also likely that the initial call to `access` and `open` would operate on the same inode. However, since there is a chance the attacker has managed to (1) `unlink` the previously `lstat`ed atom, and to (2) `symlink` it to a maze, strengthening steps are still required. The algorithm therefore continues into a K -loop that is almost identical to the one suggested by Dean and Hu (Figure 3).

```

char* chop_1st(char *path)
{
    // Find the end of the first component and
    // null-terminate it
    char *p = strchr(path, '/');

    if( p == NULL )
        return NULL;
    *p++ = '\0';

    // Handle multiple consecutive occurrences
    // of '/'. This ensures that the remainder
    // of the path is returned in a "relative"
    // form (without preceding slashes)
    for(; *p == '/'; ++p)
        ;

    // Returning NULL to indicate end of path
    return *p ? p : NULL;
}

```

Figure 6: All the parsing is encapsulated in the above function, which gets a relative path as input, chops off the first component, and returns the remainder as a relative path. (A null return value indicates the entire path was consumed and so there is no remainder.)

All the original operations are still present. The difference is that now, on each iteration, the algorithm also verifies that the atom is still a hard link. This check is necessary in order for the defense to recover, if the attacker somehow managed to win the first race and to force the algorithm into a maze while doing the **access** and **open** operations. Since the **lstat**ing of an atom is an operation that is not affected in any way by the target that a symbolic link might have, our algorithm is not vulnerable in this respect. The only other additions we have made are (1) to check that **fstat**ing the initial file we open (**fd1**) yields identical information to that pointed to by **s0**, as the *K* strengthening rounds utilize **s0** for the verification checks, and (2) to check that the **lstat**ed inode matches the initial inode, similarly to the original check with regard to the information that is retrieved by **fstat**.

Note that the two invocations of **DO_CMP** within the strengthening loop insures that all three **stat** structures are equal (**s0** = **s1** = **s2**), a check that is needed for the following reasons. By verifying that **s1** is equal to **s2**, we know for a fact that the **lstat**ed and the **opened** files are one and the same, which means we deterministically force an adversary to win a race involving a non-symlink atom, on each round. This by itself, however, is not enough, as we must also make sure that **s1** and **s2** are equal to **s0**: failing to do so would make the *K*-loop meaningless, allowing an attacker to unlawfully open the file after winning only two races, as follows

1. The attacker creates a non-symlink file, **myfile**.
2. After **is_symlink** determines that **myfile** is not a

```

int is_symlink(const char *atom,
               char target[],
               struct stat *s,
               bool *answer)
{
    int nb, l=PATH_MAX;

    DO_SYS( lstat(atom,s) );

    if( S_ISLNK(s->st_mode) ) {
        DO_SYS( nb = readlink(atom,target,l) );
        target[nb] = '\0';
        *answer = true;
    }
    else {
        *answer = false;
    }

    return 0;
}

```

Figure 7: We retrieve the name of the target file in case an atom is a symbolic link. Otherwise, the atom is a hard link in which case we record its inode information in the supplied **stat** structure for future reference. The return value indicates whether the **lstat** operations succeeded.

symlink through the **s0** stat structure, **atom_race** is invoked with **myfile** and **s0** as arguments.

3. After the initial **access** in **atom_race**, the attacker must switch **myfile** to be a symlink to the file he wishes to unlawfully access. (Race #1)
4. After the initial **open** in **atom_race**, the attacker must switch back to its original file. (Race #2)
5. All the strengthening rounds can now execute without any further effort from the attacker.

We now have everything we need in order to implement a column-oriented *K*-race traversal. The **access.open** procedure we implement does this in a straightforward manner, as is shown in Figure 9. The first chunk of code simply makes sure that the traversal is only conducted with the help of relative names (that do not start with a slash). The second chunk is the traversal per-se. This part simply iterates through the atom components, one component at a time, and takes the necessary action according to whether the atom is a symbolic link or not. The latter is the simpler alternative: if the atom is a hard link, a short **atom_race** is conducted and the atom is directly **opened**. However, if the atom is a symbolic link, the algorithm calls itself recursively to handle the newly encountered composite path. In both cases, if a valid file descriptor is returned, the algorithm is allowed to continue to the next step after **fchdir**ing to the current directory component. This strategy ensures us that there is a high probability that all relevant inodes reside in the cache during the time in which this is critical: when the *K*-race takes place.

```

int atom_race(const char *atom,
              struct stat *s0)
{
    int i, mode;
    int fd1, fd2;
    struct stat s1, s2;

    mode = S_ISDIR(s0->st_mode)
        ? X_OK /* directory */
        : R_OK /* regular */ ;

    // 1- The initial access/open
    DO_SYS( access(atom, mode) );
    DO_SYS( fd1 = open (atom, O_RDONLY) );
    DO_SYS( fstat (fd1, &s1) );
    DO_CHK( DO_CMP(s0, &s1) );

    // 2- The k strengthening rounds
    for(i=0; i<K; i++) {

        DO_SYS( lstat (atom, &s1) );
        DO_CHK( ! S_ISLNK(s1.st_mode) );
        DO_SYS( access (atom, mode) );
        DO_SYS( fd2 = open (atom, O_RDONLY) );
        DO_SYS( fstat (fd2, &s2) );

        DO_SYS( close (fd2) );
        DO_CHK( DO_CMP (s0, &s1) );
        DO_CHK( DO_CMP (s0, &s2) );
    }

    return fd1;
}

```

Figure 8: The given atom was just *lstat*ed and found to be a hard link, thus it is unlikely that an attacker would manage to set things up such that above would be thrown into a maze. If this has nevertheless happened, an additional *lstat* upon each iteration allows the algorithm to recover (compare with Figure 3).

4.1 Implementation Notes

For brevity, the presented algorithm does not handle several minor details that should be addressed in a real implementation:

First, it lacks a defense mechanism against circular symbolic links. This can be easily incorporated within the procedure shown in Figure 9 in the exact same manner as it is done within the kernel, that is, by counting the number of traversed symbolic links and aborting the procedure if the count violates some predefined threshold.

Second, our algorithm opens a file for reading only. It does not allow the caller to specify other / additional flags to be passed along to `open` (such as `O_RDWR`, `O_APPEND`, etc). There is no technical difficulty preventing us from adding a “flags” parameter that allows this, as long as we provide special treatment for file truncation (`O_TRUNC`) and forbid file creation (`O_CREAT`).

```

int access_open_2008(char *fname)
{
    int fd;
    char *suffix, target[PATH_MAX];
    struct stat s;
    bool is_sym;

    // 1- Handle the case where 'fname'
    // is an absolute path.
    if( *fname == '/' ) {
        DO_SYS( chdir("/") );
        do { ++fname; } while(*fname == '/');
        if( *fname == '\0' ) // fname is rootdir...
            return open("/", O_RDONLY);
    }

    // 2- 'fname' is now relative
    while( true ) {

        suffix = chop_1st(fname);
        DO_SYS( is_symlink(fname, target, &s, &is_sym) );

        DO_SYS( fd = (is_sym
            ? access_open_2008(target)
            : atom_race(fname, &s)) );

        if( suffix ) {
            DO_SYS( fchdir(fd) );
            DO_SYS( close (fd) );
            fname = suffix;
        }
        else
            break;
    }

    return fd;
}

```

Figure 9: A one-component-at-a-time column-oriented traversal prevents `access_open` from being abused and insures a fair atom-race is conducted when necessary. The heart of the function is the “?:” construct that decides whether to recurse over the next component (symlink) or to consume it (hard link).

Truncation is problematic as the first `open` would truncate the file regardless of whether the real user has adequate permissions to do so; the solution is to `access/open` the file without `O_TRUNC` and, if successful, to `ftruncate` the resulting descriptor. File creation raises other (independent and well-known) TOCTTOU issues that are commonly associated with the problem of creating temporary files [11]; these are outside the scope of this paper.

Additional details that should be handled are (1) setting `errno` to `EACCES` when appropriate, namely, when `DO_CMP` and `DO_CHK` fail, (2) closing already opened file descriptors (if exist) upon errors, e.g., when `fstat` fails in Figure 8, and (3) saving and restoring the working directory before and after the invocation of `access/open`, to undo the effect of using `fchdir`.

The final item raises an important point we wish to make explicit: our `access/open` implementation is inadequate for multithreaded applications if some other thread

(different than the one performing the access/open) requires the working directory to remain unchanged, as this directory is shared by all threads. We note in passing that the relatively new system call `openat` (which opens a filepath relative to a given directory file descriptor [23]) would solve this problem, as it will eliminate the need for using `fchdir`; `openat` is proposed for inclusion in the next revision of POSIX [18].

5 Crafting the Hypothetical Attack

It should come as no surprise that the new `access_open` algorithm is completely immune from the maze attack, as the latter completely lost its timing ability: the attacker colossally fails to synchronize with the activities of the defender, and has no clue about when it would be most beneficial to `unlink/link` the targeted file in order to fool the defense. Nevertheless, while we believe it is improbable, it is still possible that somebody someday would come up with some surprising approach that would allow an attacker to achieve synchronicity once again. Hence, we seek a much stronger result.

To this end, we run an experiment in which the defender is completely “exposed”: any attacker would be able to precisely know *which* actions are taken by the defender and *when*. In other words, our experiment fully reinstates the synchronicity capabilities to potential attackers, make these capabilities orders of magnitude more powerful and precise, and measures the probability attackers have to win a single round in light of the new approach; the bigger question being: Do file TOCTTOU races still pose a problem in the face of a column-oriented traversal? And if so, to what extent?

5.1 Exposed Defender

To answer this question we have implemented a defender program that provides information regarding its activities to any interested party through a shared-memory integer variable (instated with the help of SysV IPC facilities). The code of the defender is listed in Figure 10. It essentially does all of the defense-steps that are listed in Figure 8, but now each step is executed only after the defender publishes (through the shared integer) the next action to be performed. Note that the `DO_SYS` macro is redefined to record a system-call failure (instead of returning). This is done so that the defender process will not terminate. But it also means the defender maintains a fixed order of operations and thereby simplifies the code of the attacker (which is exempt from considering various corner cases). Importantly, an attacker may safely assume that the defender performs the same exact operations in the same exact order within each iteration.

In accordance to the column-oriented doctrine, the defender is operating on a file which is an atom, namely, composed of only one component that is arbitrarily called “target”. Upon each iteration, after the operation sequence is over, the defender checks whether the attack was successful, and if so increments its losses count to be printed at the end of the run. The conditions that are asserted at the *end* of each iteration are identical to those that are checked *on the fly* within Figure 8, with only one addition: the defender is made aware beforehand of the inode of the private file that the attacker wants to read; obviously, an attack is successful only if it managed to fool the defender into opening this file.

5.2 Synchronized Attacker

We now go on to review the attacker’s code, as given in Figure 11. Initially, the attacker must make sure that the file to be `lstat`d is not a symbolic link. Additionally, since the defender is going to compare the inode of the `lstat`d file to that of the `opened` file (which is the private file if the attacker gets his way), the ‘target’ file should point to the private file at this point. The attacker then waits until the defender is ready to `lstat`. As explained, the attacker’s interest dictates that the defender would be able to successfully `lstat` the private file, and so the attacker must give it enough time to do so. This is also the reason for the next ‘while’ loop that ends when the defender finishes the `lstat`, or before, depending on the heuristic we have chosen to prematurely terminate the busy-waiting: We have evaluated a wide range of $T1$ values (see next section); Note that when $T1 = 0$, the busy wait period continues until the shared variable changes. But when $T1 > 0$ waiting may be shorter, as $T1$ bounds the number of busy-wait iterations and so the smaller it is, the shorter the wait.

After the defender `lstats` the private file, the real race is on, as the defender is about to check `access` and so the attacker must arrange things such that ‘target’ will point to an appropriate location. Additionally, the attacker aspires to slow down the defender by forcing him into a maze, in order to have a better chance of winning future races. The attacker therefore `symlinks` the target to a maze. Much like with the initial `lstat` operation, the attacker must now speculate when the `access` operation is already in flight. Once again, it may be advisable to end the busy waiting before the shared variable changes, and so another timer limit – $T2$ – is employed; We allow for two different limits so as to maximize the chances of success. The attacker is now hopeful that the defender has been forced into the maze, which would mean he can safely prepare towards the next `open` by linking to the private file. But even if the attacker was not successful, this is the correct thing to do in preparation for the defender’s next `lstat` at the beginning of the next round.

```

bool sysfail;
#define DO_SYS( syscall ) \
    if( (syscall)==-1 ) \
        sysfail = true

void exposed_defender(ino_t private)
{
    struct stat s1, s2;
    int fd;

    sleep(1); // grace period for the attacker

    while( true ) {

        sysfail = false;

        *shared=LSTAT ; DO_SYS( lstat ("target", &s1 ) );
        *shared=ACCESS ; DO_SYS( access("target", R_OK ) );
        *shared=OPEN   ; DO_SYS( fd=open  ("target", O_RDONLY));
        *shared=FSTAT  ; DO_SYS( fstat (fd , &s2 ) );
        *shared=CLOSE  ; DO_SYS( close (fd ) );

        // The attacker is victorious only if all the
        // following conditions hold
        if( (! sysfail
            (! S_ISLNK(s1.st_mode) ) &&
            ( s1.st_ino == s2.st_ino ) &&
            ( s1.st_dev == s2.st_dev ) &&
            ( s2.st_ino == private ) )
            defender_loss++;
    }
}

```

Figure 10: The defender publicizes the operations about to be performed using a shared variable accessible to all.

```

void synchronized_attacker()
{
    volatile int timer1, timer2;

    unlink( "target" );
    link ( "private", "target" );

    while( true ) {

        timer1 = timer2 = 0;

        // must wait for attacker to
        // lstat private file
        while( *shared != LSTAT )
            ;

        while( *shared == LSTAT )
            if(T1 && (++timer1 >= T1))
                break;

        // now we're really racing...
        // defender is about to access
        unlink ( "target" );
        symlink( "maze", "target" );

        while( *shared == ACCESS )
            if(T2 && (++timer2 >= T2))
                break;

        unlink( "target" );
        link ( "private", "target" );
    }
}

```

Figure 11: The attacker achieves synchronicity by polling the shared variable.

6 Experimental Results

Our goal is to find out whether the column-oriented traversal technique is effective against the above hypothetical attack. (If this turns out to be the case, we can be reasonably sure that our solution would be effective in real-life scenarios where the defender is not exposed.)

6.1 Methodology

We obtain our goal by quantifying the expected time that a hypothetical attack should run in order to achieve k consecutive wins. Let this time be denoted B_k . If p is the probability for an attacker to win one round (iteration) within the exposed defender's loop, and t is the time it takes to conduct one round, then

$$B_k = t \cdot p^{-k} \quad (1)$$

because p^k is the probability for “success”, and thus, $1/p^k$ is the mean of the geometric random variable that counts the number of trials until success is observed for the first time. For example, if a round takes one millisecond ($t = 1ms$), and the probability to win a round is 1/10 ($p = 0.1$), then B_2 , B_3 , B_4 , and B_5 are 100 millisecond,

1 second, 167 minutes, and 28 hours, respectively. We approximate t and p by running the attack scenario and, upon termination, outputting (1) the duration of the attack, (2) the number of rounds conducted, and (3) the number of rounds lost. (We set t to be the average round duration, and p to be the ratio of rounds-lost to rounds-conducted.)

In order to increase the attackers' chances to win, we run the experiments on multiprocessors only. This way, attackers will have processors of their own to continuously and repeatedly attempt to fool the defender. In an effort to generalize the results, the experiments are conducted on older and recent machines, from different vendors, running different operating systems, as follows

Processor	Operating system	CPUs	Clock	Mem
UltraSPARC-II	Solaris 8	4	448 MHz	2 GB
Pentium-III	Linux 2.4.26	4	550 MHz	1 GB
Power4	AIX 5.3	8	1450 MHz	16 GB
Dual Core AMD	Linux 2.6.22	4	2200 MHz	8 GB
Intel Core 2 Duo	Linux 2.6.20	2	2400 MHz	4 GB

The ‘maze’ file we use is constructed to be the biggest that is possible on the respective OS, considering the aforementioned limits on the size of a filepath and the number of symbolic links it entails. Like Dean and Hu [12] and Borisov et al. [5] before us, we use a local file

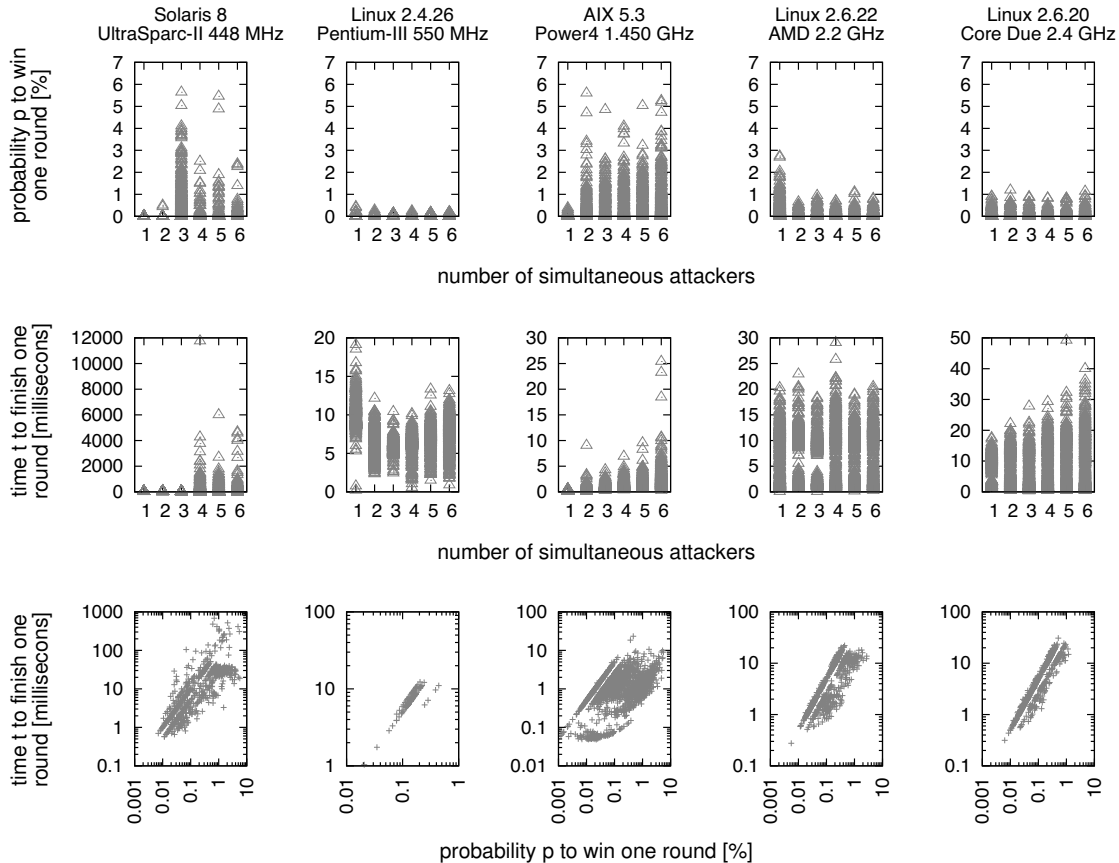


Figure 12: The probability p for a synchronized-attacker to win a single round within the loop executed by the exposed-defender (top), the time t it takes an exposed-defender to complete a single round (middle), and the connection between the two (bottom).

system for our experiments. These are the results we next describe; Afterwards, we also describe our additional findings from when running the experiments across NFS.

All the machines we use have a relatively big memory (that is, relative to the size of mazes), which as argued by Borisov et al., works against the attacker (more inodes can reside in core). However, we had appropriate permissions to change the Linux kernel running on the Pentium-III machine to one that only utilizes 256MB of the available memory. Other techniques we have experimented with in an attempt to increase the chances of the attacker to win are to simultaneously run multiple recursive `grep`-s during attacks in accordance to the suggestion by Borisov et al. [5], to launch attacks from within a huge directory that contains tens of thousands of files in accordance to Mazières and Kaashoek’s suggestion [24], and to simultaneously run several exposed-defenders on the same machine. We found that none of these techniques had a significant affect on the results, and therefore we do not report them here.

Conversely, Wei and Pu have recently shown that simultaneously running multiple identical attackers (attack-

ing the same file) on a multiprocessor system, dramatically increases the chance of a TOCTTOU attack to prevail [38]. This technique turned out to be rather successful (from the attackers’ perspective) and is therefore explicitly addressed below.

6.2 Results

Recall that the synchronized attacker has two tunable parameters — T_1 and T_2 — that place an upper bound on the two busy-wait loops the attacker must employ. We have independently set each of these two values to be either zero (no upper bound) or 2^j , where $j = 0, 1, 2, \dots, 20$. This means that we conduct 484 ($= 22^2$) experiments for any specified number of simultaneous attacker (1–6), amounting to a total of 2,904 runs, per machine.

Local FS The top of Figure 12 shows the per-machine probability (expressed as percents) for multiple simultaneous synchronized attackers to win a single round. This is plotted as a function of the number of attackers, such that each point represents one of the aforementioned 2,904

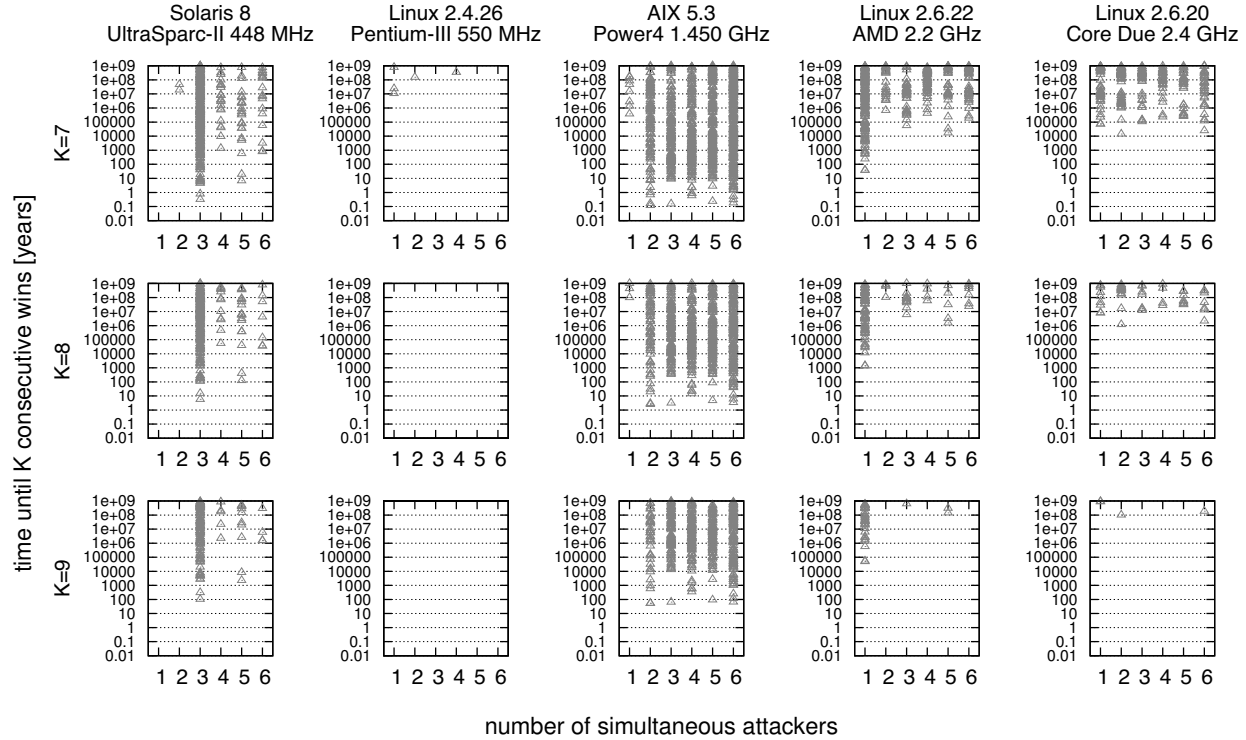


Figure 13: The expected runtime of an exposed-defender loop until k consecutive rounds are won by the attacker (B_k), for k values of 7 (top), 8 (middle), and 9 (bottom).

per-machine runs. Evidently, the probability can be quite high, culminating at nearly 6% on Sparc/Solaris (with three attackers) and on Power4/AIX (with two). Indeed, engaging more than one attacker appears beneficial, at least for these two machines.

The probability p to win a round is only one of two factors that determine the expected time B_k until a successful attack, as shown in Equation 1; The other factor is the time t it takes to complete the round, such that the bigger t is, the longer it would take to accomplish a successful attack. The middle of Figure 12 plots the values of t and shows that they too can be rather high with top values typically at tens of milliseconds, and outrageously, a few seconds in the case of Sparc/Solaris.

Importantly, the time to complete a round and the probability to win it are far from being independent variables. In fact, as shown at the bottom of Figure 12, there is a distinct linear connection between the two, which means the bigger the probability to win the round, the longer the round takes. Indeed, this makes perfect sense, as the prime objective of an attacker is to slow down the defender by throwing it into a maze. These are the two opposing side effects of the attacker's actions: maximizing p immediately translates to maximizing t , and so whatever ends up happening, the attacker inevitably contributes, to some extent, to making B_k larger.

Figure 13 assigns the t and p values of each of our experiments into Equation 1 in order to finally compute B_k , namely, the expected number of years an attack should execute until k consecutive rounds are won, for three different k values. When using $k = 7$ (the value recommended by Dean and Hu [12]) we see that a successful attack is potentially possible after a bit more than a month, in the case of Power4/AIX. Increasing k to be 8 and 9 raises the minimal expected duration to be more than 2.5 and 53 years, respectively, making the latter a safer choice in the face of our theoretical attack.

NFS Dean and Hu constrained their K -race evaluation to a local filesystem, saying that they did

“run some limited experiments attacking files across NFS and observed substantial numbers of successes. We chose not to continue these experiments, however, because NFS-accessed files are usually not the most security-critical, root privileges typically don't extend across NFS, the data displayed enormous variance depending on network and files server load.” [12]

But the set of attack experiments we conducted across NFS reveals that, while individual machines behave differently, the overall conclusion regarding the value of k does not dramatically change. The following table com-

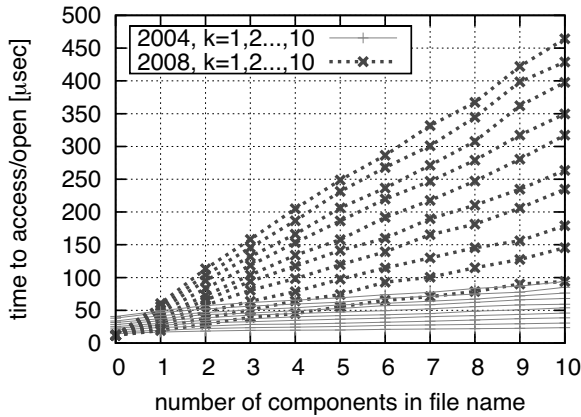


Figure 14: Overheads of `accessopen`(AMD / Linux 2.6).

compares between minimal B_k values devised when running the attack on local and a networked filesystems (each table entry is the minimal result obtained across the 2,904 respective runs; values denote years, and, if bigger than 1000, are rounded down to the closest power of ten):

Platform		Local FS			NFS		
		$k=8$	$k=9$	$k=10$	$k=8$	$k=9$	$k=10$
SPARC	Solaris 8	5.8	103	10^3	0.3	2.6	21
P-III	Linux 2.4	10^9	10^{11}	10^{13}	0.1	0.8	5.8
Power4	AIX 5.3	2.5	53	951	10^8	10^{11}	10^{13}
AMD	Linux 2.6	10^3	10^4	10^6	∞	∞	∞
Intel	Linux 2.6	10^6	10^8	10^9	9.9	129	10^3

We see that machines can become less or more vulnerable to the hypothetical attack when it is conducted across NFS. The Pentium-III machine demonstrates the most notable change, being the least susceptible to the attack within a local file system (see also Figure 13) and becoming the most vulnerable with NFS. Conversely, with the Power4 machine, it's exactly the opposite, as it transitioned from being the most vulnerable to being nearly the least, second to only the AMD machine for which no attacker wins were observed with NFS.

Robustness We note that our evaluation methodology does not constitute a proof that the proposed solution is robust. Recall, however, that the attack described here is purely hypothetical, as defenders are not likely to publish their actions through shared memory for the sake of helping attackers. We therefore argue that it is reasonable to expect that real attackers will not do better. The assumption underlying this rationale is the following: Under the newly purposed `access/open` idiom, where system calls are repeatedly applied to a single-component relative filepath, attackers will be unable to systematically and consistently slow down the defender. If this assumption is true, then our method is robust, even in the face of slow devices and multiple attackers.

Overhead Figure 14 compares the overhead of the new `access_open` to that of Dean and Hu's, as a function of the opened file's number of components. The overhead is unsurprisingly linear. Clearly the older version is faster, due to the fewer system calls it invokes. But we contend that this is tolerable, considering the older solution is unsafe and that no other portable alternative exists.

7 Generalizing

A Check-Open Utility While the above ideas were demonstrated through the `access/open` race, their applicability is broader. The maze attack is a general method to deterministically win TOCTTOU races: given a check-use pair, if an attacker can manipulate the filename being checked (or any of its components), the attacker can utilize a maze to (1) synchronize with and (2) slowdown the defender, generating the ideal conditions for the attack to succeed. Conversely, the Column-oriented K -Race (CKR) is a general method to prevent this from happening by executing the check-use pair "atomically".

Nevertheless, programmers can not be expected to tailor a CKR for every legitimate check-use scenario. We therefore aspire to devise a generic utility function that can e.g., be added to `libc`. A first immediate step is to convert our `access_open` into a `check_open` function, by allowing the caller to pass the check operation as a pointer-to-function argument (getting an atom hardlink filename and returning zero upon success.) This operation would replace the call to `access` in Figure 8, allowing programmers to pass along `access`, or `stat`, or any other conceivable filename check operation they may require.

Note that the focus on `open` as the 'use' operation is not as limited as might initially seem: Recall that bindings of file descriptors to file objects are immutable and therefore completely immune from TOCTTOU attacks. Thus, once a valid file descriptor is safely opened and returned, the programmer can securely use the wealth of system calls that operate on file descriptors (`fchown`, `fchmod`, `fchdir`, `fstat`, `ftruncate`, etc.), rather than their respective insecure TOCTTOU-prone counterparts that operate on file names (`chown`, `chmod`, `chdir`, `stat`, `truncate` etc.).

A Check-Use Utility A completely different approach would be to convert `access_open` into a general purpose `check_use` utility. Here is how such an approach might work: Hardness amplification would be removed from the core algorithm and turned into a pluggable policy to be used by programmers at will. The part that remains is a user-mode path resolution traversal. As before, the algorithm would consume one component at time, `fchdir`ing from component to component, and recursing on symlinks. The algorithm would *deterministically* make sure

it `fchdirs` to atom hard-links only (never directly to symlinks), by `lstat`ing the next atom directory (s_1), `opening` it, `fstat`ing the returned file descriptor (s_2), and making sure the s_1 and s_2 point to the same file object.

In addition to the filepath, `check_use` would get four pointer-to-function arguments F_{chk}^{dir} , F_{chk}^{link} , F_{chk}^{last} , and F_{use}^{last} . The first three are 'check' operations, respectively applied to each directory, symlink, and the last component in the given filepath, at the time the associated atom component is consumed by the path resolution traversal. Their input arguments are the atom name and the respective 'stat' structure and file descriptor (-1 for symlinks); their return value is zero to indicate the path-resolution may continue, or nonzero to indicate it should fail. The F_{use}^{last} encapsulates the 'use' operation, but otherwise has the same input and output as of the 'check' operations. All operations are invoked while the working directory of `check_use` is that of the atom that is currently being processed. Finally, the return value of `check_use` is the return value of the last operation that has failed, or that of F_{use}^{last} if all other operations succeeded.

With this design it is trivial to solve e.g., the race in Figure 2a. The garbage collector defines F_{chk}^{dir} and F_{chk}^{last} to always return 0, F_{chk}^{link} to always return -1, and F_{use}^{last} to unlink the atom file; thus, any symlink that is encountered along the way would make `check_use` fail, thereby insuring all deleted files are under the `/tmp/` directory, as required. Importantly, it does not matter whether the last (unlinked) atom is juggled by the attacker (symlink/hardlink to some sensitive file), as in this case the outcome would merely be that some link created by an attacker is deleted, a fact that does not affect the target file.

Eliminating the Probabilistic Aspect To reapply the probabilistic access/open solution under the `check_use` design, one would simply define F_{chk}^{link} to always return 0, F_{use}^{last} to return the file descriptor it gets as input, and F_{chk}^{dir} and F_{chk}^{last} to be (a slightly modified version of) `atom_race` from Figure 8. Notice, however, that there is actually no technical difficulty preventing us from going the extra mile and providing programmers with a library function that fully implements a deterministic and completely safe `access` check, in user mode: While the filepath is traversed, the associated 'stat' structure of each component, which is handed to the 'check' functions, contains the user and group ownership information as well as the user/group/world access permissions. Thus, given an arbitrary user and an atom's 'stat' structure (which is associated with an already opened file descriptor), we can deterministically decide whether the user has appropriate access permissions. While possibly a tedious task, portably implementing such a routine is nonetheless straightforward; as a library function, a single implementation would be shared by all and may have an additional benefit of

potentially being more efficient than the probabilistic approach, which involves an $O(K)$ linear loop per filepath component. We are currently in the process of evaluating this alternative (as well as the one mentioned in the following paragraph) and expect to publish the results in the near future.

Adding Credentials to the Interface In contrast to the access/open race that has a satisfactory probabilistic solution, the race depicted in Figure 2b can only be solved with the help of a deterministic user-mode `access` (as was just described), since there is no system-call equivalent to `access` that a non-setuid program can use.⁵ Indeed, defining F_{chk}^{dir} and F_{chk}^{last} to make use of the user-mode `access` and return 0 only if user "ann" has adequate permission, would suffice. Alternatively, instead of requiring the 'check' predicates to handle these details, `check_use` can be augmented to optionally get another parameter — a user id — and fail the path resolution process when an atom that the user is not allowed to open is encountered.

Summary By trading off some performance, we are able to devise a simple, yet powerful and expressive, interface that enables programmers to intuitively and securely combine a check-use pair into a single pseudo transaction, executed atomically for all practical purposes. While the entire implementation is straightforward portable user-mode, we effectively accomplish the vision of Mazières and Kaashoek (Section 2.2) regarding a new "flexible" filesystem [24]. Notably, programmers gain explicit control of whether symlinks are followed when a file is opened, and are able to specify the credentials with which relevant system calls would operate.

A facility similar to the `check_use` function suggested above, if made a standard library function, would serve three purposes. First, it will allow programmers and designers to make conscientious decisions regarding the efficiency-safety tradeoff, e.g., between insecurely opening a file with a single `open` call, or doing it in user-mode, component by component, while enforcing repeated credential checks to avoid TOCTTOU races, or maybe making the effort to develop another alternative. Second, a well-designed `check_use` facility would encapsulate the execution of vulnerable check-use pairs. When the time comes and e.g. transactional filesystems (or other relevant improvements) are made more prevalent, the internal implementation can be replaced with a more efficient alternative. Thirdly, the inclusion of a `check_use` routine in the standard API would serve educational purposes, as new programmers get familiar with the API and through it become aware of the TOCTTOU problem.

⁵An attacker can choose to link `/mail/ann` to `/etc/passwd`, rather than to `symlink`. Thus, not following symlinks will not help.

Limitations Like the maze-attack, our approach works on already-existing-files only. The TOCTTOU problem associated with creating new files (notably, when wanting to create a new temporary file [11]) is still unresolved.

8 Conclusions

The POSIX API is broken: Its semantics inherently promote TOCTTOU races between check-use operations and make systems vulnerable to malicious attacks. Existing solutions can help locate these problems, but otherwise relate to future non-prevalent systems, leaving programmers to individually come up with solutions from scratch, to numerous variants of what is provably a hard and elusive problem. We suggest to alleviate the situation by providing programmers with standard generic abstractions that effectively bind check-use pairs into a single pseudo-atomic transaction. We further show that this goal can be obtained, to a large extent, in a portable manner, in user-mode, without changing the kernel.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and to Mary Baker, the shepherd of this paper. The first author would also like to thank Nikita Borisov, Alan Hu, Ethan Miller, Wietse Venema, and Erez Zadok for providing valuable and much appreciated feedback on earlier versions of this manuscript.

References

- [1] A. Aggarwal and P. Jalote, “Monitoring the security health of software systems”. In *17th IEEE Int’l Symp. on Software Reliability Engineering (ISSRE)*, pp. 146–158, Nov 2006.
- [2] K. Ashcraft and D. Engler, “Using programmer-written compiler extensions to catch security holes”. In *IEEE Symp. on Security and Privacy (S&P)*, p. 143, May 2002.
- [3] M. Bishop, *Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux*. Technical Report CSE-95-8, University of California at Davis, Sep 1995.
- [4] M. Bishop and M. Dilger, “Checking for race conditions in file accesses”. *Computing Systems* **9**(2), pp. 131–152, Spring 1996.
- [5] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, “Fixing races for fun and profit: how to abuse atime”. In *14th USENIX Security Symp.*, pp. 303–314, Jul 2005.
- [6] D. Boulet, “UNIX domain sockets”. URL http://everything2.com/index.pl?node_id=955968, Oct 2002. (Accessed Sep 2007).
- [7] CERT Coordination Center, “CERT Advisory CA-1993-17 xterm Logging Vulnerability”. URL <http://www.cert.org/advisories/CA-1993-17.html>, Nov 1993. (Accessed Jun 2007).
- [8] H. Chen and D. Wagner, “MOPS: an infrastructure for examining security properties of software”. In *ACM Conf. on Comput. & Communi. Security (CCS)*, pp. 235–244, Nov 2002.
- [9] H. Chen, D. Wagner, and D. Dean, “Setuid demystified”. In *11th USENIX Security Symp.*, pp. 171–190, Aug 2002.
- [10] B. Chess, “Improving computer security using extended static checking”. In *IEEE Symp. on Security and Privacy (S&P)*, p. 160, May 2002.
- [11] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, “RaceGuard: kernel protection from temporary file race vulnerabilities”. In *10th USENIX Security Symp.*, pp. 165–172, Aug 2001.
- [12] D. Dean and A. J. Hu, “Fixing races for fun and profit: how to use access(2)”. In *13th USENIX Security Symp.*, pp. 195–206, Aug 2004.
- [13] D. Engler and K. Ashcraft, “RacerX: effective, static detection of race conditions and deadlocks”. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 237–252, Oct 2003.
- [14] D. Engler, B. Chelf, A. Chou, and S. Hallem, “Checking system rules using system-specific, programmer-written compiler extensions”. In *USENIX Symp. on Operating Syst. Design & Impl. (OSDI)*, p. 1, Oct 2000.
- [15] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, “Bugs as deviant behavior: a general approach to inferring errors in systems code”. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 57–72, Oct 2001.
- [16] B. Goyal, S. Sitaraman, and S. Venkatesan, “A unified approach to detect binding based race condition attacks”. In *Int’l Workshop on Cryptology & Network Security (CANS)*, Sep 2003.
- [17] A. J. Hu, “On-line publication list”. URL <http://www.cs.ubc.ca/spider/ajh/pub-list.html>. (Accessed Jun 2007).
- [18] A. Josey, “The Open Group new API set proposals”. URL http://www.opengroup.org/...austin/plato/uploads/40/9756/NAPI_overview.txt, Feb 2006. (Accessed Dec 2007).
- [19] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, “Detecting past and present intrusions through vulnerability-specific predicates”. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 91–104, Oct 2005.
- [20] C. Ko and T. Redmond, “Noninterference and intrusion detection”. In *IEEE Symp. on Security and Privacy (S&P)*, pp. 177–187, May 2002.
- [21] K-S. Lhee and S. J. Chapin, “Detection of file-based race conditions”. *Int’l J. of Information Security (IJIS)* **4**(1–2), Feb 2005.
- [22] *The access(2) manual, FreeBSD*. URL <http://www.freebsd.org/cgi/man.cgi?query=access>.

- [23] *openat(2)* — *Linux man page*. URL <http://linux.die.net/man/2/openat>.
- [24] D. Mazières and F. Kaashoek, “Secure applications need flexible operating systems”. In *IEEE Workshop on Hot Topics in Operating Syst. (HOTOS)*, p. 56, 1997.
- [25] W. S. McPhee, “Operating system integrity in OS/VS2”. *IBM Systems Journal* **13**(3), pp. 230–252, 1974. URL <http://www.research.ibm.com/journal/sj/133/ibmsj1303D.pdf>.
- [26] “National vulnerability database (NVD)”. URL <http://nvd.nist.gov/>. (Accessed Sep 2007).
- [27] J. Park, G. Lee, S. Lee, and D-K. Kim, “RPS: an extension of reference monitor to prevent race-attacks”. In 5th *Advances in Multimedia Information Processing (PCM)*, pp. 556–563, 2004. Lect. Notes Comput. Sci. vol. 3331.
- [28] C. Pu and J. Wei, “A methodical defense against TOCTTOU attacks: the EDGI approach”. In *IEEE Int’l Symp. on Secure Software Engineering (ISSSE)*, Mar 2006.
- [29] F. Schmuck and J. Wylie, “Experience with transactions in QuickSilver”. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pp. 239–253, 1991.
- [30] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West, “Model checking an entire linux distribution for security violations”. In *Ann. Comput. Security Applications Conf. (ACSAC)*, pp. 13–22, IEEE, Dec 2005.
- [31] T. Sirainen, “*fdpass.c* — File descriptor passing between processes via UNIX sockets”. URL <http://code.softwarefreedom.org/projects/backports/browser/external/standalone/dovecot/current/src/lib/fdpass.c>, 2002–2004. (Accessed Dec 2007).
- [32] W. R. Stevens and B. Fenner, *UNIX Network Programming Volume 1: The Sockets Networking API*. Addison Wesley, 3rd ed., Nov 2003. Section 15.7.
- [33] W. R. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, “RFC 3542 – advanced sockets application program interface (API) for IPv6”. URL <http://www.faqs.org/rfcs/rfc3542.html>, May 2003. (Accessed Dec 2007).
- [34] E. Tsyrlkevich and B. Yee, “Dynamic detection and prevention of race conditions in file accesses”. In 12th *USENIX Security Symp.*, pp. 243–256, Aug 2003.
- [35] P. Uppuluri, U. Joshi, and A. Ray, “Preventing race condition attacks on file-systems”. In *ACM Symp. on Applied Comput. (SAC)*, pp. 346–353, Mar 2005.
- [36] “United states computer emergency readiness team (US-CERT)”. URL <http://www.kb.cert.org/vuls>. (Accessed Sep 2007).
- [37] J. Viega, J. Bloch, Y. Kohno, and G. McGraw, “ITS4: A static vulnerability scanner for C and C++ code”. In *Ann. Comput. Security Applications Conf. (ACSAC)*, pp. 257–267, IEEE, Dec 2000.
- [38] J. Wei and C. Pu, “Multiprocessors may reduce system dependability under file-based race condition attacks”. In 37th *IEEE/IFIP Ann. Int’l Conf. on Dependable Syst. & Networks (DSN)*, Jun 2007.
- [39] J. Wei and C. Pu, “TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study.”. In 4th *USENIX Conf. on File & Storage Technologies (FAST)*, pp. 155–167, Dec 2005.
- [40] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok, “Extending ACID semantics to the file system”. *ACM Trans. on Storage (TOS)* **3**(2), p. 4, Jun 2007.
- [41] A. C. Yao, “Theory and applications of trapdoor functions”. In 23rd *IEEE Symp. on Foundations of Computer Science*, pp. 80–91, 1982.
- [42] K. Zeilenga, H. Chu, and P. Masarati, “*libraries/libutil/getpeereuid.c*”. OpenLDAP source code URL <http://www.openldap.org/devel/cvsweb.cgi>, 2000–2007. (Accessed Dec 2007).

EIO: Error Handling is Occasionally Correct

Haryadi S. Gunawi, Cindy Rubio-González,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Ben Liblit
Computer Sciences Department, University of Wisconsin-Madison

Abstract

The reliability of file systems depends in part on how well they propagate errors. We develop a static analysis technique, EDP, that analyzes how file systems and storage device drivers propagate error codes. Running our EDP analysis on all file systems and 3 major storage device drivers in Linux 2.6, we find that errors are often incorrectly propagated; 1153 calls (13%) drop an error code without handling it.

We perform a set of analyses to rank the robustness of each subsystem based on the completeness of its error propagation; we find that many popular file systems are less robust than other available choices. We confirm that write errors are neglected more often than read errors. We also find that many violations are not corner-case mistakes, but perhaps intentional choices. Finally, we show that inter-module calls play a part in incorrect error propagation, but that chained propagations do not. In conclusion, error propagation appears complex and hard to perform correctly in modern systems.

1 Introduction

The robustness of file systems and storage systems is a major concern, and rightly so [32]. Recent work has shown that file systems are especially unreliable when the underlying disk system does not behave as expected [20]. Specifically, many modern commodity file systems, such as Linux ext3 [31], ReiserFS [23], IBM's JFS [1], and Windows NTFS [27], all have serious bugs and inconsistencies in how they handle errors from the storage system. However, the question remains unanswered as to why these fault-handling bugs are present.

In this paper, we investigate what we believe is one of the root causes of deficient fault handling: *incorrect error code propagation*. To be properly handled, a low-level error code (e.g., an “I/O error” returned from a device driver) must be correctly propagated to the appropriate code in the file system. Further, if the file system is unable to recover from the fault, it may wish to pass the error up to the application, again requiring correct error propagation.

Without correct error propagation, any comprehensive failure policy is useless: recovery mechanisms and policies cannot be invoked if the error is not propagated. Incorrect error propagation has been a significant problem in many systems. For example, self-healing systems cannot heal themselves if error signals never reach the self-recovery modules [6, 26], components behind an interface do not receive error notifications [16], and distributed systems often obtain misleading error codes [15, 30], which turns into frustration for human debugging. In summary, if errors are not propagated, then the effort spent detecting and recovering from those errors [4, 5, 18, 21, 22, 28, 29] is worthless.

To analyze how errors are propagated in file and storage system code, we have developed a static source-code analysis technique. Our technique, named *Error Detection and Propagation (EDP)* analysis, shows how error codes flow through the file system and storage drivers. EDP performs a dataflow analysis by constructing a function-call graph showing how error codes propagate through return values and function parameters.

We have applied EDP analysis to all file systems and 3 major storage device drivers (SCSI, IDE, and Software RAID) implemented in Linux 2.6. We find that *error handling is occasionally correct*. Specifically, we see that low-level errors are sometimes lost as they travel through the many layers of the storage subsystem: out of the 9022 function calls through which the analyzed error codes propagate, we find that 1153 calls (13%) do not correctly save the propagated error codes.

Our detailed analysis enables us to make a number of conclusions. First, we find that the more complex the file system (in terms of both lines of code and number of function calls with error codes), the more likely it is to incorrectly propagate errors; thus, these more complex file systems are more likely to suffer from silent failures. Second, we observe that I/O write operations are more likely to neglect error codes than I/O read operations. Third, we find that many violations are not corner-case mistakes: the return codes of some functions are consistently ignored, which makes us suspect that the omissions are intentional. Finally, we show how inter-module calls play a major part in causing incorrect error propagation, but that chained propagations do not.

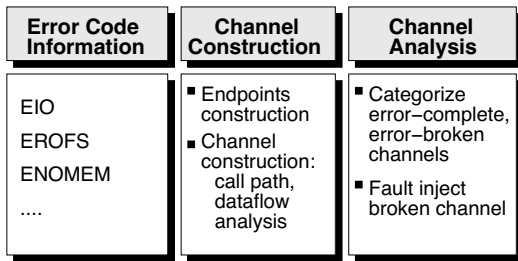


Figure 1: **EDP Architecture.** The diagram shows the framework for Error Detection and Propagation (EDP) analysis of file and storage systems code.

The rest of this paper is organized as follows. We describe our methodology and present our results in Section 2 and 3 respectively. To understand the root causes of the problem, we perform a set of deeper analyses in Section 4. Section 5 and 6 discuss future work and related work respectively. Finally, Section 7 concludes.

2 Methodology

To understand the propagation of error codes, we have developed a static analysis technique that we name *Error Detection and Propagation (EDP)*. In this section, we identify the components of Linux 2.6 that we will analyze and describe EDP.

2.1 Target Systems

In this paper, we analyze how errors are propagated through the file systems and storage device drivers in Linux 2.6.15.4. We examine all Linux implementations of file systems that are located in 51 directories. These file systems are of different types, including disk-based file systems, network file systems, file system protocols, and many others. Our analysis follows requests through the virtual file system and memory management layers as well. In addition to file systems, we also examine three major storage device drivers (SCSI, IDE, and software RAID), as well as all lower-level drivers. Beyond these subsystems, our tool can be used to analyze other Linux components as well.

2.2 EDP Analysis

The basic mechanism of EDP is a dataflow analysis: EDP constructs a function-call graph covering all cases in which error codes propagate through return values or function parameters. To build EDP, we harness C Intermediate Language (CIL) [19]. CIL performs source-to-source transformation of C programs and thus can be used in the analysis of large complex programs such as the Linux kernel. The EDP analysis is written as a CIL extension in 4000 lines of code in the OCaml language.

Subsystem	Single (seconds)	Full (seconds)	Subsystem Size (Kloc)
VFS	4	—	34
Mem. Mgmt.	3	—	20
XFS	8	13	71
ReiserFS	3	8	24
ext3	2	7	12
Apple HFS	1	6	5
VFAT	1	5	1
All File Systems Together	47		372

Table 1: **EDP Performance.** The table shows the EDP runtime for different subsystems. “Single” runtime represents the time to analyze each subsystem in isolation without interaction with other subsystems (e.g., VFS and MM). “Full” runtime represents the time to analyze a file system along with the virtual file system and the memory management. The last row reports the time to analyze all of the file systems together.

The abstraction that we introduce in EDP is that error codes flow along *channels*, where a channel is the set of function calls between where an error code is first generated and where it is terminated (e.g., by being either handled or dropped). As shown in Figure 1, EDP contains three major components. The first component identifies the error codes that will be tracked. The second constructs the channels along which the error codes propagate. Finally, the third component analyzes the channels and classifies each as being either complete or broken.

Table 1 reports the EDP runtime for different subsystems, running on a machine with 2.4 GHz Intel Pentium 4 CPU and 512 MB of memory. Overall, EDP analysis is fast; analyzing all file systems together in a single run only takes 47 seconds. We now describe the three components of EDP in more detail.

2.2.1 Error Code Information

The first component of EDP identifies the error codes to track. One example is EIO, a generic error code that commonly indicates I/O failure and is used extensively throughout the file system; for example, in ext3, EIO touches 266 functions and propagates through 467 calls. Besides EIO, many kernel subsystems commonly use other error codes as defined in `include/asm-generic/errno.h`. In total, there are hundreds of error codes that are used for different purposes. We report our findings on the propagation of 34 basic error codes that are mostly used across all file systems and storage device drivers. These error codes can be found in `include/asm-generic/errno-base.h`.

2.2.2 Channel Construction

The second component of EDP constructs the *channel* in which the specified error codes propagate. A channel can be constructed from function calls and asynchronous wake-up paths; in our current analysis, we focus only on function calls and discuss asynchronous paths in Section 5.3.

We define a channel by its two endpoints: generation and termination. The *generation endpoint* is the function that exposes an error code, either directly through a return value (e.g., the function contains a `return -EIO` statement) or indirectly through a function argument passed by reference. After finding all generation endpoints, EDP marks each function that propagates the error codes; *propagating functions* receive error codes from the functions that they call and then simply propagate them in a return value or function parameter. The *termination endpoint* is the function in which an error code is no longer propagated in the return value or a parameter of the function.

One of the major challenges we address when constructing error channels is handling function pointers. The typical approach for handling function pointers is to implement a points-to analysis [13] that identifies the set of real functions each function pointer might point at; however, field-sensitive points-to analyses can be expensive. Therefore, we customize our points-to analysis to exploit the systematic structure that these pointers exhibit.

First, we keep track of all structures that have function pointers. For example, the VFS read and write interfaces are defined as fields in the `file_ops` structure:

```
struct file_ops {
    int (*read) ();
    int (*write) ();
};
```

Since each file system needs to define its own `file_ops`, we automatically find all global instances of such structures, look for the function pointer assignments within the instances, and map function-pointer implementations to the function pointer interfaces. For example, `ext2` and `ext3` define their file operations like this:

```
struct file_ops ext2_f_ops {
    .read = ext2_read;
    .write = ext2_write;
};
struct file_ops ext3_f_ops {
    .read = ext3_read;
    .write = ext3_write;
};
```

Given such global structure instances, we add the interface implementations (e.g., `ext2_read`) to the implementation list of the corresponding interfaces (e.g.,

`file_ops->read`). Although this technique connects most of the mappings, a function pointer assignment could still occur in an instruction rather than in a global structure instance. Thus, our tool also visits all functions and finds any assignment that maps an implementation to an interface. For example, if we find an assignment such as `f_op->read = ntfs_read`, then we add `ntfs_read` to the list of `file_ops->read` implementations.

In the last phase, we change function pointer calls to direct calls. For example, if VFS makes an interface call such as `(f_op->read)()`, then we automatically rewrite such an assignment to:

```
switch (...) {
    case ext2:  ext2_read(); break;
    case ext3:  ext3_read(); break;
    case ntfs:  ntfs_read(); break;
    ...
}
```

Across all Linux file systems and storage device drivers, there are 191 structural interfaces (e.g., `file_ops`), 904 function pointer fields (e.g., `read`), 5039 implementations (e.g., `ext2_read`), and 2685 function pointer calls (e.g., `(f_op->read)()`). Out of 2865 function pointer calls, we connect all except 564 calls (20%). The unconnected 20% of calls are due to indirect implementation assignment. For example, we cannot map assignment such as `f_op->read = f`, where `f` is either a local variable or a function parameter, and not a function name. While it is feasible to traceback such assignments using stronger and more expensive analysis, we assume that major interfaces linking modules together have already been connected as part of global instances. If all calls are connected, more error propagation chain can be analyzed, which means more violations are likely to be found.

2.2.3 Channel Analysis

The third component of EDP distinguishes two kinds of channels: error-complete and error-broken channels. An *error-complete* channel is a channel that minimally checks the occurrence of an error. An error-complete channel thus has this property at its termination endpoint:

$$\exists \text{ if } (expr) \{ \dots \}, \text{ where } \text{errorCodeVariable} \subseteq expr$$

which states that an error code is considered checked if there exist an `if` condition whose expression contains the variable that stores the error code. For example, the function `goodTerminationEndpoint` in the code segment below carries an error-complete channel because the function saves the returned error code (line 2) and checks the error code (line 3):

```

1 void goodTerminationEndpoint() {
2     int err = generationEndpoint();
3     if (err)
4         ...
5 }
6 int generationEndpoint() {
7     return -EIO;
8 }

```

Note that an error could be checked but not handled properly, *e.g.* no error handling in the `if` condition. Since error handling is usually specific to each file system, and hence there are many instances of it, we decided to be “generous” in the way we define how error is handled, *i.e.* by just checking it. More violations might be found when we incorporate all instances of error handling.

An *error-broken* channel is the inverse of an error-complete channel. In particular, the error code is either *unsaved*, *unchecked*, or *overwritten*. For example, the function `badTerminationEndpoint` below carries an error-broken channel of unchecked type because the function saves the returned error code (line 2) but it never checks the error before the function exits (line 3):

```

1 void badTerminationEndpoint() {
2     int err = generationEndpoint();
3     return;
4 }

```

An error-broken channel is a serious file system bug because it can lead to a silent failure. In a few cases, we inject faults in error-broken channels to confirm the existence of silent failures. We utilize our block-level fault injection technique [20] to exercise error-broken channels that relate to disk I/O. In a broken channel, we look for two pieces of information: which workload and which failure led us to that channel. After finding the necessary information, we run the workload, inject the specific block failure, and observe the I/O traces and the returned error codes received in upper layers (*e.g.*, the application layer) to confirm whether a broken channel leads to a silent failure. The reader will note that our fault-injection technique is limited to disk I/O related channels. To exercise all error-broken channels, techniques such as symbolic execution and directed testing [9, 10] that simulate the environment of the component in test would be of great utility.

2.2.4 Limitations

Error propagation has complex characteristics: correct error codes must be returned; each subsystem uses both generic and specific error codes; one error code could be mapped to another; error codes are stored not only in scalar variables but also in structures (*e.g.*, control blocks); and error codes flow not only through function calls but also asynchronously via interrupts and callbacks. In our static analysis, we have not modeled all

these characteristics. Nevertheless, by just focusing on the propagation of basic error codes via function call, we have found numerous violations that need to be fixed. A more complete tool that covers the properties above would uncover even more incorrect error handling.

3 Results

We have performed EDP analysis on all file systems and storage device drivers in Linux 2.6.15.4. Our analysis studies how 34 basic error codes (*e.g.*, `EIO` and `ENOMEM`) defined in `include/asm-generic/errno-base.h` propagate through these subsystems. We examine these basic error codes because they involve thousands of functions and propagate across thousands of calls.

In these results, we distinguish two types of violations that make up an error-broken channel: *unsaved* and *unchecked* error codes (overwritten codes have been deferred to future work; see Section 5.1 for more information). An *unsaved error code* is found when a callee propagates an error code via the return value, but the caller does not save the return value (*i.e.*, it is treated as a void-returning call even though it actually returns an error code). Throughout the paper, we refer to this type of broken channel as a “*bad call*.” An *unchecked error code* is found when a variable that may contain an error code is neither checked nor used in the future; we always refer to this case as an unchecked code.

3.1 Unsaved Error Codes

First, we report the number of error-broken channels due to a caller simply not saving the returned error code (*i.e.*, the number of bad calls). The simplified HFS code below shows an example of unsaved error code. The function `find_init` accepts a new uninitialized `find_data` structure (line 2), allocates a memory space for the `search_key` field (line 3), and returns `ENOMEM` error code when the memory allocation fails (line 5). However, one of its callers, `file_lookup`, does not save the returned error code (line 10) but tries to access the `search_key` field which still points to `NULL` (line 11). Hence, a null-pointer dereference takes place and the system could crash or corrupt data.

```

1 // hfs/bfind.c
2 int find_init(find_data *fd) {
3     fd->search_key = kmalloc(...)
4     if (!fd->search_key)
5         return -ENOMEM;
6     ...
7 }
8 // hfs/inode.c
9 int file_lookup() {
10     find_init(fd); /* NOT-MADE E.C */
11     fd->search_key->cat = ...; /* BAD!! */
12     ...
13 }

```

Viol#	Caller → Callee		Filename	Line#
A	file_lookup	find_init	inode.c	493
B	fill_super	find_init	super.c	385
C	lookup	find_init	dir.c	30
D	brec_updt_pnt	__brec_find	brec.c	405
E	brec_updt_pnt	__brec_find	brec.c	345
F	cat_delete	free_fork	catalog.c	228
G	cat_delete	find_init	catalog.c	213
H	cat_create	find_init	catalog.c	95
I	file_trunc	free_exts	extent.c	507
J	file_trunc	free_exts	extent.c	497
K	file_trunc	find_init	extent.c	494
L	ext_write_ext	find_init	extent.c	135
M	ext_read_ext	find_init	extent.c	188
N	brec_rmv	__brec_find	brec.c	193
O	readdir	find_init	dir.c	68
P	cat_move	find_init	catalog.c	280
Q	brec_insert	__brec_find	brec.c	145
R	free_fork	free_exts	extent.c	307
S	free_fork	find_init	extent.c	301

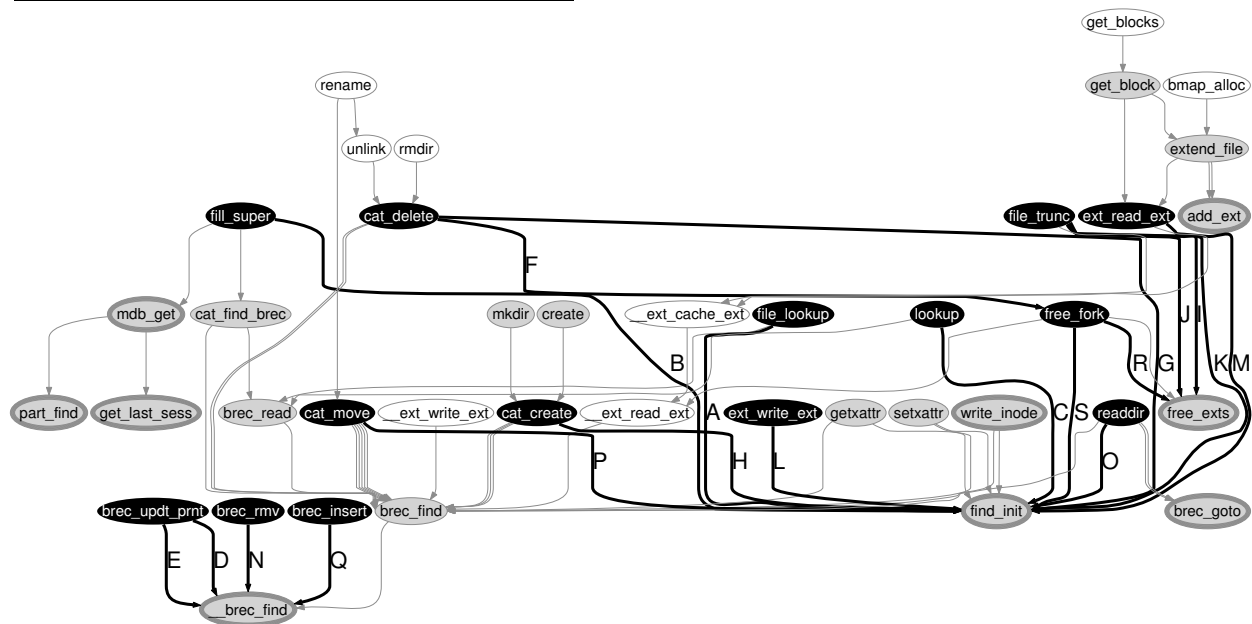
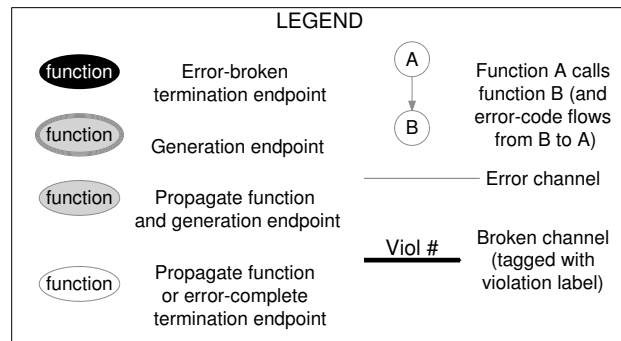
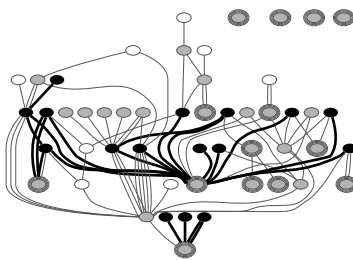
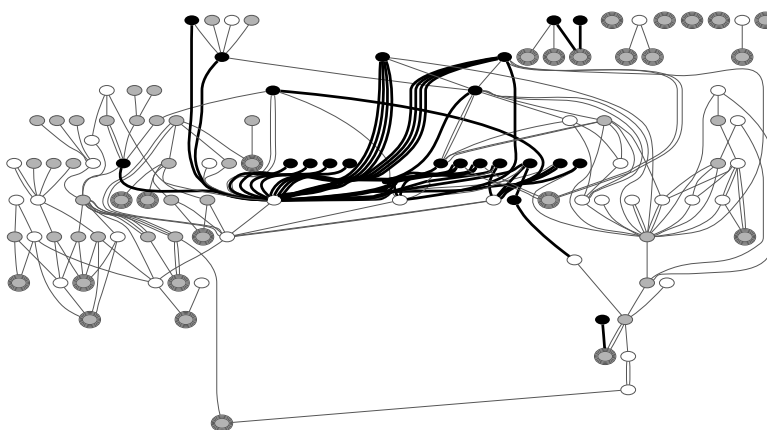


Figure 2: A Sample of EDP Output. The lower figure depicts the EDP output for the HFS file system. Some function names have been shortened to improve readability. As summarized in the upper right legend, a gray node with a thicker border represents a function that generates an error code. The other gray node represents the same thing, but the function also propagates the error code received from its callee. A white node represents a good function, i.e. it either propagates the error code to its caller or if it does not propagate the error code it minimally checks the error code. A black node represents an error-broken termination endpoint, i.e. it is a function that commits the violation of unsaved error codes. The darker and thicker edge coming out from a black node implies a broken error channel (a bad call); an error code actually flows from its callee, but the caller drops the error code. For ease of debugging, each bad call is labeled with a violation number whose detailed information can be found in the upper left violation table. For example, violation #E found in the bottom left corner of the graph is a bad call made by `brec_updt_pnt` when calling `__brec_find`, which can be located in `fs/hfs/brec.c` line 345.

HFS+ [22 bad / 84 calls, 26%]



ext3 [37 bad / 188 calls, 20%]



ReiserFS [35 bad / 218 calls, 16%]

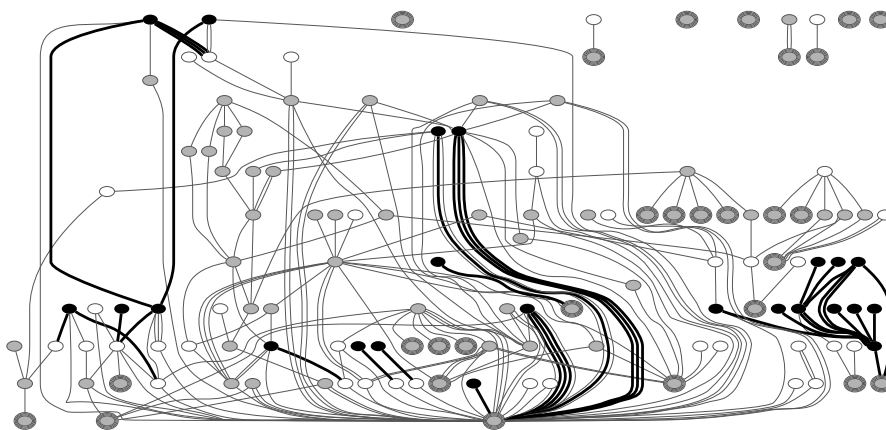
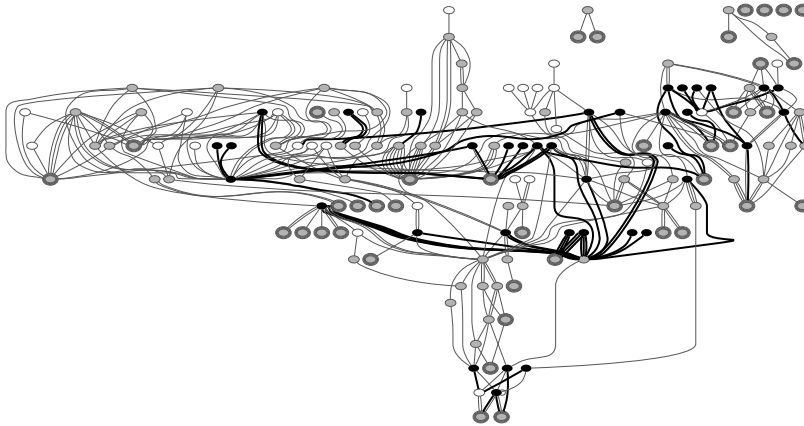
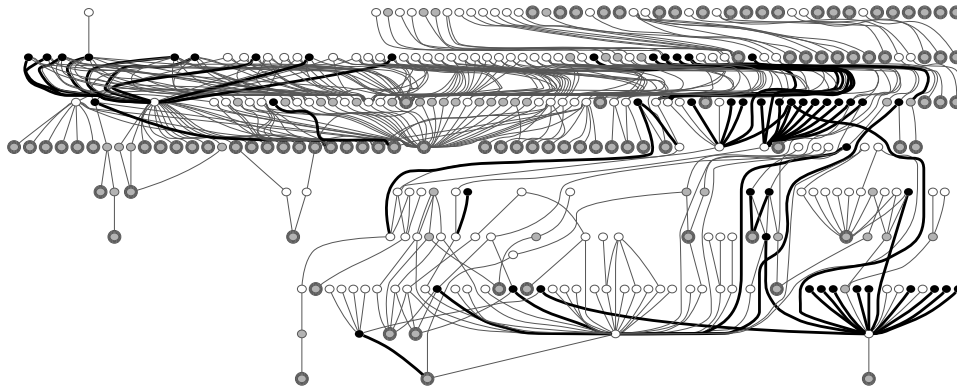


Figure 3: More Samples of EDP Output. The figures illustrate the prevalent problem of incomplete error-propagation across different types of file systems. Details such as function names and violation numbers have been removed. Gray edges represent calls that propagate error codes. Black edges represent bad calls. The number of edges are reported in [X / Y , $Z\%$] format where X and Y represent the number of black and all (gray and black) edges respectively, and Z represents the fraction of X and Y . For more information, please see the legend in Figure 2.

IBM JFS [61 bad / 340 calls, 18%]



NFS Client [54 bad / 446 calls, 12%]



XFS [105 bad / 1453 calls, 7%]

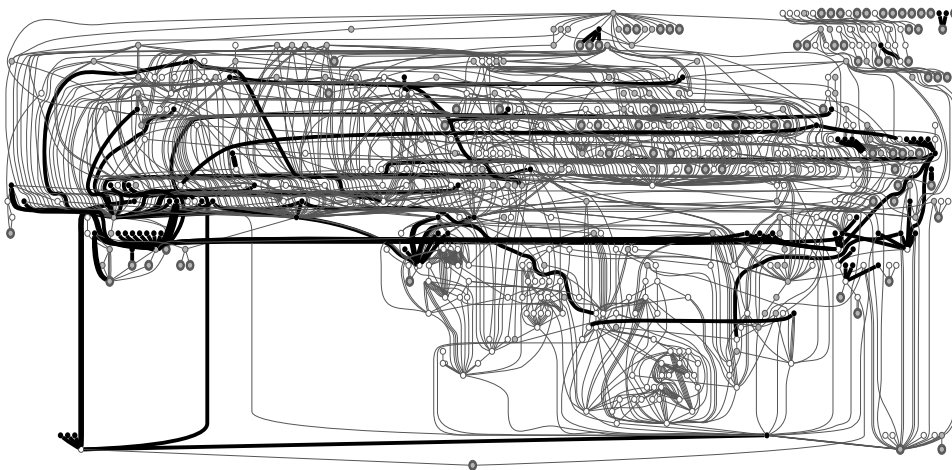


Figure 4: More Samples of EDP Output (Cont'd). Please see caption in Figure 3.

File Systems

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
XFS	101	1457	71	6.9	1.4
Virtual FS	96	1149	34	8.4	2.9
IBM JFS	95	390	17	24.4	5.6
ext3	80	362	12	22.1	7.2
NFS Client	62	482	18	12.9	3.6
CIFS	43	339	21	12.7	2.1
ReiserFS	42	399	24	10.5	1.8
Mem. Mgmt.	40	351	20	11.4	2.0
Apple HFS+	25	98	7	25.5	3.7
JFFS v2	24	153	11	15.7	2.2
Apple HFS	20	76	5	26.3	4.8
SMB	19	196	6	9.7	3.5
ext2	18	103	6	17.5	3.3
AFS	16	62	7	25.8	2.6
NTFS	15	186	18	8.1	0.9
NFS Server	15	265	14	5.7	1.2
NCP	13	169	5	7.7	2.6
UFS	12	44	5	27.3	2.6
JBD	10	43	4	23.3	2.6
FAT	9	81	4	11.1	2.9
Plan 9	9	80	4	11.2	2.4
System V	7	30	3	23.3	3.2
JFFS	7	56	5	12.5	1.4
UDF	6	50	9	12.0	0.7
MSDOS	5	39	1	12.8	9.3
VFAT	4	39	1	10.3	5.0
Minix	4	31	4	12.9	1.2

File Systems (Cont'd)

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
FUSE	4	48	3	8.3	1.5
Automounter4	4	53	2	7.5	2.7
NFS Lockd	3	21	4	14.3	0.8
Relayfs	2	5	1	40.0	2.7
Partitions	2	3	4	66.7	0.6
ISO	2	19	3	10.5	0.7
HugeTLB Sup	2	10	1	20.0	3.0
Compr. ROM	2	3	1	66.7	4.5
ADFS	2	30	2	6.7	1.3
sysfs sup.	1	29	2	3.4	0.8
romfs sup.	1	3	1	33.3	2.4
ramfs sup.	1	6	1	16.7	6.0
QNX 4	1	8	2	12.5	0.9
proc fs sup.	1	44	6	2.3	0.2
OS/2 HPFS	1	18	6	5.6	0.2
FreeVxFS	1	4	2	25.0	0.7
EFS	1	3	1	33.3	1.4
devpts	1	2	1	50.0	6.2
Boot FS	1	9	1	11.1	1.2
BeOS	1	5	3	20.0	0.5
Automounter	1	41	2	2.4	1.0
Amiga FFS	1	34	3	2.9	0.3
exportfs sup.	0	1	1	0.0	0.0
Coda	0	149	3	0.0	0.0
Total	833	7278	366	—	—
Average	16.3	142.7	7.2	17.0	2.4

Storage Drivers

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
SCSI (root)	123	628	198	19.6	0.6
IDE (root)	53	223	15	23.8	3.5
Block Dev (root)	39	195	36	20.0	1.1
Software RAID	31	290	32	10.7	1.0
SCSI (aacraid)	30	76	7	39.5	4.8
SCSI (lpfc)	14	30	16	46.7	0.9
Blk Dev (P-IDE)	11	17	8	64.7	1.5
SCSI aic7xxx	8	62	37	12.9	0.2
IDE (pci)	5	106	12	4.7	0.4

Storage Drivers (Cont'd)

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
IDE legacy	2	3	3	66.7	0.8
Blk Layer Core	2	65	8	3.1	0.3
SCSI megaraid	1	30	6	3.3	0.2
Blk Dev (Eth)	1	5	2	20.0	0.7
SCSI (sym53c8)	0	6	10	0.0	0.0
SCSI (qla2xxx)	0	8	49	0.0	0.0
Total	320	1744	430	—	—
Average	21.3	116.3	28.6	22.4	1.1

Table 2: **Error-broken channels due to unsaved error codes.** These tables report the number of bad calls found across all file systems and storage device drivers in Linux 2.6.15.4. In each table, from left to right column we report the name of the subsystem, the number of bad calls, the number of error channels (i.e., the number of calls to functions that propagate error codes), the size of the subsystem, the fraction of bad calls over all error-related calls (ratio of 2nd and 3rd column), and finally the number of violations per Kloc (ratio of 2nd and 4th column). We categorize a directory as a subsystem. Thus, for storage drivers, since different SCSI device drivers exist in the first-level of the `scsi/` directory, we put all of them as one subsystem. SCSI device drivers that are located in different directories (e.g., `scsi/lpfc/`, `scsi/aacraid/`) are categorized as different subsystems. The same principle is applied to IDE.

To show how EDP is useful in finding error propagation bugs, we begin by showing a sample of EDP analysis for a simple file system, Apple HFS. Then, we present our findings on all subsystems that we analyze, and finally discuss false positives.

3.1.1 EDP on Apple HFS

Figure 2 depicts the EDP output when analyzing the propagation of the 34 basic error codes in the Apple HFS file system. There are two important elements that EDP produces in order to ease the debugging process. First, EDP generates an error propagation graph that only includes functions and function calls through which the analyzed error codes propagate. From the graph, one can easily catch all bad calls and functions that make the bad calls. Second, EDP provides a table that presents more detailed information for each bad call (*e.g.*, the location where the bad call is made).

Using the information that EDP provides, we found three major error-handling inconsistencies in HFS. First, 11 out of 14 calls to `find_init` drop the returned error codes. As described earlier in this section, this bug could cause the system to crash or corrupt data. Second, 4 out of 5 total calls to the function `_brec_find` are bad calls (as indicated by the four black edges, E, D, N, and Q, found in the lower left of the graph). The task of this function is to find a record in an HFS node that best matches the given key, and return `ENOENT` (no entry) error code if it fails. The only call that saves this error code is made by the wrapper, `brec_find`. Interestingly, all 18 calls to this wrapper propagate the error code properly (as indicated by all gray edges coming into the function).

Finally, 3 out of 4 calls to `free_exts` do not save the returned error code (labeled R, I, and J). This function traverses a list of extents and locates the extents to be freed. If the extents cannot be found, the function returns `EIO`. More interestingly, the developer wrote a comment “panic?” just before the return statement (maybe in the hope that in this failure case the callers will call panic, which will never happen if the error code is dropped). By and large, we found similar inconsistencies in all the subsystems we analyzed. The fact that the fraction of bad calls over all calls to a function is generally high is intriguing, and will be discussed further in Section 4.3.

3.1.2 EDP on All File Systems and Storage Drivers

Figure 3 and 4 show EDP outputs for six more file systems whose error-propagation graphs represent an interesting sample. EDP outputs for the rest of the file systems can be downloaded from our web site [11]. A small file system such as HFS+ has simple propagation chains, yet bad calls are still made. More complex error propagation can be seen in `ext3`, `ReiserFS`, and `IBM JFS`; within

these file systems, error-codes propagate throughout 180 to 340 function calls. The error propagation in `NFS` is more structured compared to other file systems. Finally, among all file systems we analyze, `XFS` has the most complex error propagation chain; almost 1500 function calls propagate error-codes. Note that each graph in Figures 3 and 4 was produced by analyzing each file system in isolation (*i.e.*, the graph only shows intra-module but not inter-module calls), yet they already illustrate the complexity of error code propagation in each file system. Manual code inspection would require a tremendous amount of work to find error-propagation bugs.

Next, we analyzed the propagation of error codes across all file systems and storage device drivers as a whole. All inter-module calls were connected by our EDP channel constructor, which connects all function pointer calls; hence, we were able to catch inter-module bad calls in addition to intra-module ones. Table 2 summarizes our findings. Note that the number of violations reported is higher than the ones reported in Figures 2, 3, and 4 because we catch more bugs when we analyze each file system in conjunction with other subsystems (*e.g.*, `ext3` with the journaling layer, `VFS`, and the memory management).

Surprisingly, out of 9022 error channels, 1153 (or nearly 13%) constitute bad calls. This appears to be a long-standing problem. We ran a partial analysis in Linux 2.4 (not shown) and found that the magnitude of incomplete error code propagation is essentially the same. In Section 4, we try to dissect the root causes of this problem.

3.1.3 False Positives

It is important to note that while the number of bad calls is high, not all bad calls could cause damage to the system. The primary reason is what we call a *double error code*; some functions expose two or more error codes at the same time, and checking one of the error codes while ignoring the others can still be correct. For example, in the `ReiserFS` code below, the error code returned from `sync_dirty_buffer` does not have to be saved (line 8) *if and only if* the function performs the check on the second error code (line 9); the buffer must be checked whether it is up-to-date.

```
1 // fs/buffer.c
2 int sync_dirty_buffer (buffer_head* bh) {
3     ...
4     return ret; // RETURN ERROR CODE
5 }
6 // reiserfs/journal.c
7 int flush_commit_list() {
8     sync_dirty_buffer(bh); // UNSAVED EC
9     if (!buffer_uptodate(bh)) {
10         return -EIO;
11     }
12 }
```

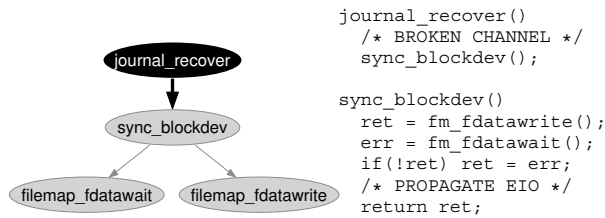


Figure 5: **Silent error in journal recovery.** In the figure on the left, EDP marks `journal_recover` as a termination endpoint of a broken channel. The code snippet on the right shows that `journal_recover` ignores the EIO propagated by `sync_blockdev`.

To ensure that the number of false positives we report is not overly large, we manually analyze all of the code snippets to check whether a second error code is being checked. Note that this manual process can be automated if we incorporate all types of error codes into EDP. We have found only a total of 39 false positives, which have been excluded from the numbers we report in this paper. Thus, the high numbers in Table 2 provide a hint to a real and critical problem.

3.2 Silent Failures: Manifestations of Unsaved Error Codes

To show that unsaved error codes represent a serious problem that can lead to silent failures, we injected disk block failures in a few cases. As shown in Figure 5, one serious silent failure arises during file system recovery: the journaling block device layer (JBD) does not properly propagate any block write failures, including inode, directory, bitmap, superblock, and other block write failures. EDP unearths these silent failures by pinpointing the `journal_recover` function, which is responsible for file system recovery, as it calls `sync_blockdev` to flush the dirty buffer pages owned by the block device. Unfortunately, `journal_recover` does not save the error code propagated by `sync_blockdev` in the case of block write failures. This is an example where the error code is dropped in the middle of its propagation chain; `sync_blockdev` correctly propagates the EIO error codes received from the two function calls it makes.

A similar problem occurs in the NFS server code. From a similar failure injection experiment, we found that the NFS client is not informed when a write failure occurs during a sync operation. In the experiment, the client updates old data and then sends a sync operation with the data to the NFS server. The NFS server then invokes the `nfstdosync` operation, which mainly performs three operations similar to the `sync_blockdev` call above. First, the NFS server writes dirty pages to the disk; second, it writes dirty inodes and the superblock to disk; third, it waits until the ongoing I/O data transfer

terminates. All these three operations could return error codes, but the implementation of `nfstdosync` does not save any return values. As a result, the NFS client will never notice any disk write failures occurring in the server. Thus, even a careful, error-robust client cannot trust the server to inform it of errors that occur.

In the NFS server code, we might expect that at least one return value would be saved and checked properly. However, no return values are saved, leading one to question whether the returned error codes from the `write` or `sync` operations are correctly handled in general. It could be the case that the developers are not concerned about write failures. We investigate this hypothesis in Section 4.2.

3.3 Unchecked Error Code

Lastly, we report the number of error-broken channels due to a variable that contains an error code not being checked or used in the future. For example, in the IBM JFS code below, `rc` carries an error code propagated from `txCommit` (line 4), but `rc` is never checked.

```

1 // jfs/jfs_txnmgr.c
2 int jfs_sync () {
3     int rc;
4     rc = txCommit(); // UNCHECKED 'rc'
5     // No usage or check of 'rc'
6     // after this line
7 }

```

This analysis can also report false positives due to the double error code problem described previously. In addition, we also find the problem of *overloaded variables* that contribute as false positives. We define a variable to be overloaded if the variable could contain an error code or a data value. For instance, `blknum` in the QNX4 code below is an example of an overloaded variable:

```

1 // qnx4/dir.c
2 int qnx4_readdir () {
3     int blknum;
4     struct buffer_head *bh;
5     blknum = qnx4_block_map();
6     bh = sb_bread (blknum);
7     if (bh == NULL)
8         // error
9 }

```

In this code, `qnx4_block_map` could return an error code (line 5), which is usually a negative value. `sb_bread` takes a block number and returns a buffer head that contains the data for that particular block (line 6). Since a negative block number will lead to a `NULL` buffer head (line 7), the error code stored in `blknum` does not have to be explicitly checked. The developer believes that the other part of the code will catch this error or eventually raise related errors. This practice reduces the accuracy of our static analysis.

Rank	By % Broken		By Viol/Kloc	
	FS	Frac.	FS	Viol/Kloc
1	IBM JFS	24.4	ext3	7.2
2	ext3	22.1	IBM JFS	5.6
3	JFFS v2	15.7	NFS Client	3.6
4	NFS Client	12.9	VFS	2.9
5	CIFS	12.7	JFFS v2	2.2
6	MemMgmt	11.4	CIFS	2.1
7	ReiserFS	10.5	MemMgmt	2.0
8	VFS	8.4	ReiserFS	1.8
9	NTFS	8.1	XFS	1.4
10	XFS	6.9	NFS Server	1.2

Table 3: Least Robust File Systems. *The table shows the ten least robust file systems using two ranking systems. In the first ranking system, file system robustness is ranked based on the fraction of broken channels over all error channels (the 5th column of Table 2). The second ranking system sorts file systems based on the number of broken channels found in every Kloc (the 6th column of Table 2).*

Since the number of unchecked error code reports is small, we were able to remove the false positives and find a total of 3 and 2 unchecked error codes in file systems and storage drivers, respectively, that could lead to silent failures.

4 Analysis of Results

In the following sections, we present five analyses whereby we try to uncover the root causes and impact of incomplete error propagation. Since the number of unchecked and overwritten error codes is small, we only consider unsaved error codes (bad calls) in our analyses; thus we use “bad calls” and “broken channels” interchangeably from now on. First, we made a correlation between robustness and complexity. Second, we analyzed whether file systems and storage device drivers give different treatment to errors occurring in I/O read vs. I/O write operations. From that analysis we find that many write errors are neglected; hence we perform the next study in which we try to answer whether ignored errors are corner-case mistakes or intentional choices. In the final two analyses, we analyze whether chained error propagation and inter-module calls play major parts in causing incorrect error propagation.

4.1 Complexity and Robustness

In our first analysis, we would like to correlate the number of mistakes in a subsystem with the complexity of that subsystem. For file systems, XFS with 71 Kloc has more mistakes than other, smaller file systems. However, it is not necessary that XFS is seen as the least robust file system. Table 3 sorts the robustness of each file system

based on two rankings. In both rankings, we only account file systems that are at least 10 Kloc in size with at least 50 error-related calls, *i.e.* we only consider “complex” file systems.

A noteworthy observation is that ext3 and IBM JFS are ranked as the two least robust file systems. This fact affirms our earlier findings on the robustness of ext3 and IBM JFS [20]. In this prior work, we found that ext3 and IBM JFS are inconsistent in dealing with different kinds of disk failures. Thus, it might be the case that these inconsistent policies correlate with inconsistent error propagation.

Among storage device drivers, it is interesting to compare the robustness of the SCSI and IDE subsystems. If we compare SCSI and IDE subsystems using the first ranking system, SCSI and IDE are almost comparable (21% vs. 18%). However, if we compare them based on the second ranking system, then the SCSI subsystem is almost four times more robust than IDE (0.6 vs. 2.1 errors/Kloc). Nevertheless it seems the case that SCSI utilizes basic error codes much more than IDE does.

When the robustness of storage drivers and file systems is compared using the first ranking, on average storage drivers are less robust compared to file systems (22% vs. 17%, as reported in the last rows of Table 2). On the other hand, in the second ranking system, storage drivers are more robust compared to file systems (1.1 vs. 2.4 mistakes/Kloc). From our point of view, the first ranking system is more valid because a subsystem could be comprised of submodules that do not necessarily use error codes; what is more important is the number of bad calls in the population of all error-related calls.

4.2 Neglected Write Errors

As mentioned in Section 3.2, we have observed that error codes propagated in `write` or `sync` operations are often ignored. Thus, we investigate how many write errors are neglected compared to read errors. This study is motivated by our findings in that section as well as by our earlier findings where we found that at least for ext3, read failures are detected, but write errors are often ignored [20].

To perform this study, we filter out calls that do not relate to read and write operations. Since it is impractical to do that manually, we use a simple string comparison to mark calls that are relevant to our analysis. That is we only take a caller→callee pair where the callee contains the string `read`, `write`, `sync`, or `wait`. We include `wait`-type calls because in many cases `wait`-type callees (*e.g.*, `filemap_datawait`) represent waiting for one or more I/O operations and could return error information on the operation. Thus, in our study, `write`-, `sync`-, and `wait`-type calls are categorized as write operations.

Callee Type	Bad Calls	EC Calls	Frac. (%)
Read*	26	603	4.3
Sync	70	236	29.7
Wait	27	70	38.6
Write	80	598	13.4
Sync+Wait+Write	177	904	19.6
Specific Callee			
filemap_fdatawait	22	29	75.9
filemap_fdatawrite	30	47	63.8
sync_blockdev	15	21	71.4

Table 4: **Neglected write errors in file system code.**

The table shows that read errors are handled more correctly than write errors. The upper table shows the fraction of bad calls over four category of calls: read, sync, wait, and write. The later three can be categorized as a write operation. The lower table shows neglected write errors for three specific functions. The 29 (*) violated read calls are all related to readahead and asynchronous read; in other words, all error codes returned in synchronous reads are being saved and checked.

The upper half of Table 4 reports our findings. The last column shows how often errors are ignored in the file system code. Interestingly, file systems have a tendency to correctly handle error codes propagated from read-type calls, but not those from write-type calls (4.3% vs. 19.6%). The 29 (4.3%) unsaved read error codes are all found in readahead operations in the memory management subsystem; it might be acceptable to ignore prefetch read errors because such reads can be reissued in the future whenever the page is actually read.

As discussed in Section 3.1, a function could return more than one error code at the same time, and checking only one of them suffices. However, if we know that a certain function only returns a single error code and yet the caller does not save the return value properly, then we would know that such call is really a flaw. To find real flaws in the file system code, we examined three important functions that we know only return single error codes: `sync_blockdev`, `filemap_fdatawrite`, and `filemap_fdatawait`. A file system that does not check the returned error codes from these functions would obviously let failures go unnoticed in the upper layers.

The lower half of Table 4 reports our findings. Many error codes returned from the three methods are simply not saved ($> 63\%$ in all cases). Two conclusions might be drawn from this observation. First, this could suggest that higher-level recovery code does not exist (since if it exists, it will not be invoked due to the broken error channel), or it could be the case that errors are intentionally neglected. We consider this second possibility in greater detail in the next section.

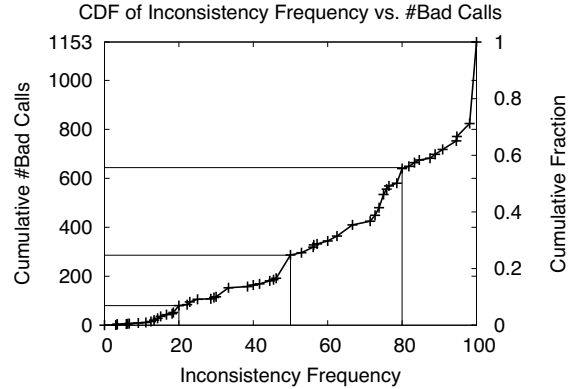


Figure 6: **Inconsistent calls frequency.** The figure shows that inconsistent calls are not corner-case bugs. The x-axis represents the inconsistent-call frequency of a function. $x=20\%$ means that there is one bad call out of five total calls; $x=80\%$ means that there are four bad calls out of five total calls. The left y-axis counts the cumulative number of bad calls. For example, below the 20% mark, there are 80 bad calls that have an inconsistent-call frequency of less than 20%. As reported in Table 2, there exist a total of 1153 bad calls. The right y-axis shows the cumulative fraction of bad calls over the 1153 bad calls.

4.3 Inconsistent Calls: Corner Case or Majority?

In this section, we consider the nature of *inconsistent* calls. For example, we found that 1 out of 33 calls to `ide_setup_pci_device` does not save the return value. One would probably consider this single call as an inconsistent implementation because the majority of the calls to that function save the return value. On the other hand, we also found that 53 out of 54 calls to `unregister_filesystem` do not save the return error codes. Assuming that most kernel developers are essentially competent, this suggests that it may actually be safe to not check the error code returned from this particular function.

To quantify inconsistent calls, we define the *inconsistent call frequency* of a function as the ratio of bad calls over all error-related calls to the function, and correlate this frequency with the number of bad calls to the function. For example, the inconsistent call frequencies for `ide_setup_pci_blockdev` and `unregister_filesystem` are 3% (1/33) and 98% (53/54) respectively and the numbers of bad calls are 1 and 53 respectively.

Figure 6 plots the cumulative distribution function of this behavior. The graph could be seen as a means to prioritize which bad calls to fix first. Bad calls that fall below the 20% mark could be treated as *corner cases*, i.e. we should be suspicious on one bad call in the midst

of four good calls to the same function. On the other hand, bad calls that fall above the 80% mark could hint that either different developers make the same mistake and ignore it, or it is probably safe to make such a mistake.

One perplexing phenomenon visible in the graph is that around 871 bad calls fall above the 50% mark. In other words, they cannot be considered as corner-case bugs; the developers might be aware of these bad calls, but probably just ignore them. One thing we have learned from our recent work on file system code is that if a file system does not know how to recover from a failure, it has the tendency to just ignore the error code. For example, ext3 ignores write failures during checkpointing simply because it has no recovery mechanism (e.g., chained transactions [12]) to deal with such failures. Thus, we suspect that there are deeper design shortcomings behind poor error code handling; error code mismanagement may be as much symptom as disease.

Our analysis is similar to the work of Engler *et al.* on findings bugs automatically [8]. In their work, they use existing implementation to imply beliefs and facts. Applying their analysis to our case, the bad calls that fall above the 80% mark might be considered as good calls. However, since we are analyzing the specific problem of error propagation, we use that semantic knowledge and demand a discipline that promotes checking an error code in all circumstances, rather than one that follows majority rules.

4.4 Characteristics of Error Channels

Finally, we study whether the characteristic of an error channel has an impact on the robustness of error code propagation in that channel. In particular, we explore two characteristics of error channels: one based on the error propagation distance and one based on the location distance (inter- vs. intra-file calls).

With the first characteristic, we would like to find out whether error codes are lost near the generation endpoint or somewhere in the middle of the propagation chain. We distinguish two calls: direct-error and propagate-error calls. In a *direct-error call*, the callee is an error-generation endpoint. In a *propagate-error call*, the callee is not a generation endpoint; rather it is a function that propagates an error code from one of the functions that it calls, i.e. it is a function in the middle of the propagation chain. Next, we define a *bad* direct-error (or propagate-error) call as a direct-error (or propagate-error) call that does not save the returned error code.

Initially, we assumed that the frequency of bad propagate-error calls would be higher than that of bad direct-error calls; we assumed error codes tend to be dropped in the middle of the chain rather than near the generation endpoint. It turns out that the number of bad

	Bad Calls	EC Calls	Frac. (%)
<i>File Systems</i>			
Inter-module	307	1944	15.8
Inter-file	367	2786	13.2
Intra-file	159	2548	6.2
<i>Storage Drivers</i>			
Inter-module	48	199	24.1
Inter-file	92	495	18.6
Intra-file	180	1050	17.1

Table 5: **Calls based on location distance.** *The table shows that the fraction of bad calls in inter-module calls is higher than the one in inter-file calls. Similarly, inter-file calls are less robust than intra-file calls. Note that “inter-file” refers to cross-file calls within the same module. Inter-file calls across different modules are categorized as inter-module.*

direct-error and propagate-error calls are similar for file system code but the other way around for storage driver code. In particular, for file systems, the ratio of bad over all direct-error calls is 10%, and the ratio of bad over all propagate-error calls is 14%. For storage drivers, they are 20% and 15% respectively.

Lastly, in the second characteristic, we categorized calls based on the location distance between a caller and a callee. In particular, we distinguish three calls: inter-module, inter-file (but within the same module), and intra-file calls. Table 5 reports that intra-file calls are more robust than inter-file calls, and inter-file calls are more robust than intra-file calls. For example, out of 1944 inter-module calls in which error codes propagate in file system, 307 (16%) of them are bad calls. However, out of 2786 inter-file calls within the same module, there are only 367 (13%) bad calls. Intra-file calls only exhibit 6% bad calls. The same pattern occurs in storage device drivers. Thus, we conclude that the location distance between the caller and the callee plays a role in the robustness of the call.

5 Future Work

In this section, we discuss some of the issues we previously deferred regarding how to build complete and accurate static error propagation analysis. In general, we plan to refine our static analysis with the intention of uncovering more violations within the file and storage system stack.

5.1 Overwritten Error Codes

In this paper, we examined broken channels that are caused by unsaved and unchecked error codes; broken channels can also be caused by *overwritten error codes*,

in which the container that holds the error code is overwritten with another value before the previous error is checked. For example, the CIFS code below overwrites (line 6) the previous error code received from another call (line 4).

```
1 // cifs/transport.c
2 int SendReceive () {
3     int rc;
4     rc = cifs_sign_smb(); // PROPAGATE E.C.
5     ... // No use of 'rc' here
6     rc = smb_send(); // OVERWRITTEN
7 }
```

Currently, EDP detects overwritten error codes, but reports too many false positives to be useful. We are in the process of fine-tuning EDP so that it provides more accurate output. The biggest problem we have encountered is due to the nature of the error hierarchy: in many cases, a less critical error code is overwritten with a more critical one. For example, in the memory management code below, when first encountering a page error, the error code is set to `EIO` (line 6). Later, the function checks whether the flags of a `map` structure carry a no-space error code (line 8). If so, the `EIO` error code is overwritten (line 9) with a new error code `ENOSPC`.

```
1 // mm/filemap.c
2 int wait_on_page_writeback_range (pg, map) {
3     int ret = 0;
4     ...
5     if (PageError(pg))
6         ret = -EIO;
7     ...
8     if (test_bit(AS_ENOSPC, &map->flags))
9         ret = -ENOSPC;
10    if (test_bit(AS_EIO, &map->flags))
11        ret = -EIO;
12    return ret;
13 }
```

Manually inspecting the results obtained from EDP, we have identified five real cases of overwritten error codes: one each in AFS and FAT, and three in CIFS. We believe we will find more cases as we fine-tune our analysis of overwritten error codes.

5.2 Error Transformation

Our current EDP analysis focuses on the basic error codes that are stored and propagated mainly in integer containers. However, file and storage systems also use other specific error codes stored in complex structures that can be mapped to other error codes in new error containers; we call this issue *error transformation*. For example, the block layer clears the `uptodate` bit stored in a buffer structure to signal I/O failure, while the VFS layer simply uses generic error codes such as `EIO` and `EROFS`. We have observed a path where an error container changes five times, involving four different types

of containers. A complete EDP analysis must recognize all transformations. With a more complete analysis, we expect to see even more violations.

5.3 Asynchronous Error Channels

Finally, we plan to expand our definition of error channels to include *asynchronous paths*. We briefly describe two examples of asynchronous paths and their complexities. First, when a lower layer interrupts an upper one to notify it of the completion of an I/O, the low-level I/O error code is usually stored in a structure located in the heap; the receiver of the interrupt should grab the structure and check the error it carries, but tracking this propagation through the heap is not straightforward. Another example occurs during journaling: a journal daemon is woken up somewhere in the `fsync()` path and propagates a journal error code via a global journal state. When we consider asynchronous error channels, we also expect the number of violations to increase.

6 Related Work

Previous work has used static techniques to understand variety of problems in software systems. For example, Meta-level compilation (MC) [7, 8] enables a programmer to write simple, system-specific compiler extensions to automatically check software for rule violations. With their work, one can find broken channels by specifying a rule such as “a returned variable must be checked.” Compared to their work, ours presents more information on how error propagates and convert it into graphical output for ease of analysis and debugging.

Another related project is FiSC [32], which uses the model-checking tool CMC [17] to find file system errors in the Linux kernel. Every time the file system under test transitions to a new state, FiSC runs a series of invariant checkers looking for file system errors. If an error is found, one can trace back the states and diagnose the sequence of actions that lead to the error. One aspect of our work that is similar to FiSC is that we unearth silent failures. For example, FiSC detects a bug where a system call returns success after it calls a resource allocation routine that fails, *e.g.* due to memory failures.

In recent work, Johansson analyzes run-time error propagation based on interface observations [14]. Specifically, an error is injected at the OS-driver interface by changing the value of a data parameter. By observing the application-OS interface after the error injection, they reveal whether errors occurring in the OS environment (device drivers) will propagate through the OS and affect applications. This run-time technique is complementary to our work, especially to uncover the eventual bad effects of error-broken channels.

Solving the error propagation problem is also similar to solving the problem of unchecked exceptions. Sacramento *et al.* found too many unchecked exceptions, thus doubting programmers' assurances in documenting exceptions [25]. Nevertheless, since using exceptions is not a kernel programming style, at least at the current state, solutions to the problem of unchecked exceptions might not be applicable to kernel code. Only recently is there an effort in employing exceptions in OS code [3].

Our tool is also similar to Jex [24]. While Jex is a static analysis tool that determines exception flow information in Java programs, our tool determines the error code flow information within the Linux kernel.

To fix the incomplete error propagation problem, developers could simply adopt a simple set-check-use methodology [2]. However, it is interesting to see that this simple practice has not been applied thoroughly in file systems and storage device drivers. As mentioned in Section 4.3, we suspect that there are deeper design shortcomings behind poor error code handling.

7 Conclusion

In this paper, we have analyzed the file and storage systems in Linux 2.6 and found that error codes are not consistently propagated. We conclude by reprinting some developer comments we found near some problematic cases:

CIFS – “*Not much we can do if it fails anyway, ignore rc.*”

CIFS – “*Should we pass any errors back?*”

ext3 – “*Error, skip block and hope for the best.*”

ext3 – “*There's no way of reporting error returned from ext3_mark_inode_dirty() to userspace. So ignore it.*”

IBM JFS – “*Note: todo: log error handler.*”

ReiserFS – “*We can't do anything about an error here.*”

XFS – “*Just ignore errors at this point. There is nothing we can do except to try to keep going.*”

SCSI – “*Retval ignored?*”

SCSI – “*Todo: handle failure.*”

These comments from developers indicate part of the problem: even when the developers are aware they are not properly propagating an error, they do not know how to implement the correct response. Given static analysis tools to identify the source of bugs (such as EDP), developers may still not be able to fix all bugs in a straightforward manner.

Due to these observations, we believe it is thus time to rethink how failures are managed in large systems.

Preaching that developers follow error handling conventions and hoping the resulting systems work as desired seems naive at best. New approaches to error detection, propagation, and recovery are needed; in the future, we plan to explore a range of error architectures, hoping to find methods that increase the level of robustness in the storage systems upon which we all rely.

Acknowledgments

We thank the members of the ADSL research group for their insightful comments. We would also like to thank Geoff Kuenning (our shepherd) and the anonymous reviewers for their excellent feedback and comments, many of which have greatly improved this paper. The second author wishes to thank the National Council on Science and Technology of Mexico and the Secretariat of Public Education for their financial support.

This work is supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

- [1] Steve Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.
- [2] Michael W. Bigrigg and Jacob J. Vos. The Set-Check-Use Methodology for Detecting Error Propagation Failures in I/O Routines. In *WDB '02*, Washington, DC, June 2002.
- [3] Bruno Cabral and Paulo Marques. Making Exception Handling Work. In *HotDep II*, Seattle, Washington, Nov 2006.
- [4] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI '04*, pages 31–44, San Francisco, CA, December 2004.
- [5] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX '98 Security*, San Antonio, TX, January 1998.
- [6] Daniel Ellard and James Megquier. DISP: Practical, Efficient, Secure, and Fault-Tolerant Distributed Data Storage. *ACM Transactions on Storage (TOS)*, 1(1):71–94, Feb 2005.
- [7] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallen. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *OSDI '00*, San Diego, CA, October 2000.

- [8] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*, pages 57–72, Banff, Canada, October 2001.
- [9] Dawson R. Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA '07*, London, United Kingdom, July 2007.
- [10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI '05*, Chicago, IL, June 2005.
- [11] Haryadi S. Gunawi. EDP Output for All File Systems. www.cs.wisc.edu/adsl/Publications/eio-fast08/readme.html.
- [12] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherd. In *SOSP '07*, pages 283–296, Stevenson, Washington, October 2007.
- [13] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *PASTE '01*, Snowbird, Utah, June 2001.
- [14] Andreas Johansson and Neeraj Suri. Error Propagation Profiling of Operating Systems. In *DSN '05*, Yokohoma, Japan, June 2005.
- [15] George Kola, Tevfik Kosar, and Miron Livny. Faults in Large Distributed Systems and What We Can Do About Them. In *Euro-Par*, August 2005.
- [16] Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *FTCS-29*, Madison, Wisconsin, June 1999.
- [17] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *OSDI '02*, Boston, MA, December 2002.
- [18] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.
- [19] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *CC '02*, pages 213–228, April 2002.
- [20] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.
- [21] Feng Qin, Shan Lu, and Yuanyuan Zhou. Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA-11*, San Francisco, California, February 2005.
- [22] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *SOSP '05*, Brighton, UK, October 2005.
- [23] Hans Reiser. ReiserFS. www.namesys.com, 2004.
- [24] Martin P. Robillard and Gail C. Murphy. Designing Robust Java Programs with Exceptions. In *FSE '00*, San Diego, CA, November 2000.
- [25] Paulo Sacramento, Bruno Cabral, and Paulo Marques. Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions? In *IVNET '06*, Florianopolis, Brazil, October 2006.
- [26] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *USENIX '05*, Anaheim, CA, April 2005.
- [27] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.
- [28] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*, Bolton Landing, NY, October 2003.
- [29] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI '04*, pages 1–16, San Francisco, CA, December 2004.
- [30] Douglas Thain and Miron Livny. Error Scope on a Computational Grid: Theory and Practice. In *HPDC 11*, Edinburgh, Scotland, July 2002.
- [31] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.
- [32] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.

An Analysis of Data Corruption in the Storage Stack

Lakshmi N. Bairavasundaram*, Garth R. Goodson†, Bianca Schroeder‡

Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*

**University of Wisconsin-Madison* †*Network Appliance, Inc.* ‡*University of Toronto*
{laksh, dusseau, remzi}@cs.wisc.edu, garth.goodson@netapp.com, bianca@cs.toronto.edu

Abstract

An important threat to reliable storage of data is silent data corruption. In order to develop suitable protection mechanisms against data corruption, it is essential to understand its characteristics. In this paper, we present the first large-scale study of data corruption. We analyze corruption instances recorded in production storage systems containing a total of 1.53 million disk drives, over a period of 41 months. We study three classes of corruption: checksum mismatches, identity discrepancies, and parity inconsistencies. We focus on checksum mismatches since they occur the most.

We find more than 400,000 instances of checksum mismatches over the 41-month period. We find many interesting trends among these instances including: (i) nearline disks (and their adapters) develop checksum mismatches an order of magnitude more often than enterprise class disk drives, (ii) checksum mismatches within the same disk are not independent events and they show high spatial and temporal locality, and (iii) checksum mismatches across different disks in the same storage system are not independent. We use our observations to derive lessons for corruption-proof system design.

1 Introduction

One of the biggest challenges in designing storage systems is providing the reliability and availability that users expect. Once their data is stored, users expect it to be persistent forever, and perpetually available. Unfortunately, in practice there are a number of problems that, if not dealt with, can cause data loss in storage systems.

One primary cause of data loss is disk drive unreliability [16]. It is well-known that hard drives are mechanical, moving devices that can suffer from mechanical problems leading to drive failure and data loss. For example, media imperfections, and loose particles causing scratches, contribute to media errors, referred to as

latent sector errors, within disk drives [18]. Latent sector errors are detected by a drive's internal error-correcting codes (ECC) and are reported to the storage system.

Less well-known, however, is that current hard drives and controllers consist of hundreds-of-thousands of lines of low-level firmware code. This firmware code, along with higher-level system software, has the potential for harboring bugs that can cause a more insidious type of disk error – silent data corruption, where the data is silently corrupted with no indication from the drive that an error has occurred.

Silent data corruptions could lead to data loss more often than latent sector errors, since, unlike latent sector errors, they cannot be detected or repaired by the disk drive itself. Detecting and recovering from data corruption requires protection techniques beyond those provided by the disk drive. In fact, basic protection schemes such as RAID [13] may also be unable to detect these problems.

The most common technique used in storage systems to detect data corruption is for the storage system to add its own higher-level checksum for each disk block, which is validated on each disk block read. There is a long history of enterprise-class storage systems, including ours, in using checksums in a variety of manners to detect data corruption [3, 6, 8, 22]. However, as we discuss later, checksums do not protect against all forms of corruption. Therefore, in addition to checksums, our storage system also uses file system-level disk block identity information to detect previously undetectable corruptions.

In order to further improve on techniques to handle corruption, we need to develop a thorough understanding of data corruption characteristics. While recent studies provide information on whole disk failures [11, 14, 16] and latent sector errors [2] that can aid system designers in handling these error conditions, very little is known about data corruption, its prevalence and its characteristics. This paper presents a large-scale study of silent data corruption based on field data from 1.53 million disk drives covering a time period of 41 months. We use the

same data set as the one used in recent studies of latent sector errors [2] and disk failures [11]. We identify the fraction of disks that develop corruption, examine factors that might affect the prevalence of corruption, such as disk class and age, and study characteristics of corruption, such as spatial and temporal locality. To the best of our knowledge, this is the first study of silent data corruption in production and development systems.

We classify data corruption into three categories based on how it is discovered: checksum mismatches, identity discrepancies, and parity inconsistencies (described in detail in Section 2.3). We focus on checksum mismatches since they are found to occur the most. Our important observations include the following:

- (i) During the 41-month time period, we observe more than 400,000 instances of checksum mismatches, 8% of which were discovered during RAID reconstruction, creating the possibility of real data loss. Even though the rate of corruption is small, the discovery of checksum mismatches during reconstruction illustrates that data corruption is a real problem that needs to be taken into account by storage system designers.
- (ii) We find that nearline (SATA) disks and their adapters develop checksum mismatches an order of magnitude more often than enterprise class (FC) disks. Surprisingly, enterprise class disks with checksum mismatches develop more of them than nearline disks with mismatches.
- (iii) Checksum mismatches are not independent occurrences – both within a disk and within different disks in the same storage system.
- (iv) Checksum mismatches have tremendous spatial locality; on disks with multiple mismatches, it is often consecutive blocks that are affected.
- (v) Identity discrepancies and parity inconsistencies do occur, but affect 3 to 10 times fewer disks than checksum mismatches affect.

The rest of the paper is structured as follows. Section 2 presents the overall architecture of the storage systems used for the study and Section 3 discusses the methodology used. Section 4 presents the results of our analysis of checksum mismatches, and Section 5 presents the results for identity discrepancies, and parity inconsistencies. Section 6 provides an anecdotal discussion of corruption, developing insights for corruption-proof storage system design. Section 7 presents related work and Section 8 provides a summary of the paper.

2 Storage System Architecture

The data we analyze is from tens-of-thousands of production and development Network ApplianceTM storage systems (henceforth called *the system*) installed at hun-

dreds of customer sites. This section describes the architecture of the system, its corruption detection mechanisms, and the classes of corruptions in our study.

2.1 Storage Stack

Physically, the system is composed of a storage-controller that contains the CPU, memory, network interfaces, and storage adapters. The storage-controller is connected to a set of disk shelves via Fibre Channel loops. The disk shelves house individual disk drives. The disks may either be enterprise class FC disk drives or nearline serial ATA (SATA) disks. Nearline drives use hardware adapters to convert the SATA interface to the Fibre Channel protocol. Thus, the storage-controller views all drives as being Fibre Channel (however, for the purposes of the study, we can still identify whether a drive is SATA and FC using its model type).

The software stack on the storage-controller is composed of the WAFL[®] file system, RAID, and storage layers. The file system processes client requests by issuing read and write operations to the RAID layer, which transforms the file system requests into logical disk block requests and issues them to the storage layer. The RAID layer also generates parity for writes and reconstructs data after failures. The storage layer is a set of customized device drivers that communicate with physical disks using the SCSI command set [23].

2.2 Corruption Detection Mechanisms

The system, like other commercial storage systems, is designed to handle a wide range of disk-related errors. The data integrity checks in place are designed to detect and recover from corruption errors so that they are not propagated to the user. The system does not knowingly propagate corrupt data to the user under any circumstance.

We focus on techniques used to detect silent data corruption, that is, corruptions not detected by the disk drive or any other hardware component. Therefore, we do not describe techniques used for other errors, such as transport corruptions reported as SCSI transport errors or latent sector errors. Latent sector errors are caused by physical problems within the disk drive, such as media scratches, “high-fly” writes, etc. [2, 18], and detected by the disk drive itself by its inability to read or write sectors, or through its error-correction codes (ECC).

In order to detect silent data corruptions, the system stores extra information to disk blocks. It also periodically reads all disk blocks to perform data integrity checks. We now describe these techniques in detail.

Corruption Class	Possible Causes	Detection Mechanism	Detection Operation
Checksum mismatch	Bit-level corruption; torn write; misdirected write	RAID block checksum	Any disk read
Identity discrepancy	Lost or misdirected write	File system-level block identity	File system read
Parity inconsistency	Memory corruption; lost write; bad parity calculation	RAID parity mismatch	Data scrub

Table 1: Corruption classes summary.

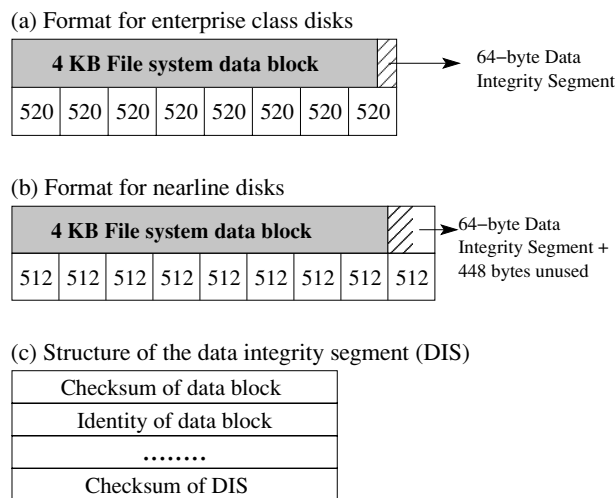


Figure 1: **Data Integrity Segment.** The figure shows the different on-disk formats used to store the data integrity segment of a disk block on (a) enterprise class drives with 520B sectors, and on (b) nearline drives with 512B sectors. The figure also shows (c) the structure of the data integrity segment. In particular, in addition to the checksum and identity information, this structure also contains a checksum of itself.

2.2.1 Data Integrity Segment

In order to detect disk block corruptions, the system writes a 64-byte data integrity segment along with each disk block. Figure 1 shows two techniques for storing this extra information, and also describes its structure. For enterprise class disks, the system uses 520-byte sectors. Thus, a 4-KB file system block is stored along with 64 bytes of data integrity segment in eight 520-byte sectors. For nearline disks, the system uses the default 512-byte sectors and store the data integrity segment for each set of eight sectors in the following sector. We find that the protection offered by the data integrity segment is well-worth the extra space needed to store them.

One component of the data integrity segment is a checksum of the entire 4 KB file system block. The checksum is validated by the RAID layer whenever the data is read. Once a corruption has been detected, the

original block can usually be restored through RAID reconstruction. We refer to corruptions detected by RAID-level checksum validation as *checksum mismatches*.

A second component of the data integrity segment is block identity information. In this case, the fact that the file system is part of the storage system is utilized. The identity is the disk block's identity within the file system (e.g., this block belongs to inode 5 at offset 100). This identity is cross-checked at file read time to ensure that the block being read belongs to the file being accessed. If, on file read, the identity does not match, the data is reconstructed from parity. We refer to corruptions that are not detected by checksums, but detected through file system identity validation as *identity discrepancies*.

2.2.2 Data Scrubbing

In order to pro-actively detect errors, the RAID layer periodically *scrubs* all disks. A data scrub issues read operations for each physical disk block, computes a checksum over its data, and compares the computed checksum to the checksum located in its data integrity segment. If the checksum comparison fails (i.e., a checksum mismatch), the data is reconstructed from other disks in the RAID group, after those checksums are also verified. If no reconstruction is necessary, the parity of the data blocks is generated and compared with the parity stored in the parity block. If the parity does not match the verified data, the scrub process fixes the parity by regenerating it from the data blocks. In a system protected by double parity, it is possible to definitively tell which of the parity or data block is corrupt.

We refer to these cases of mismatch between data and parity as *parity inconsistencies*. Note that data scrubs are unable to validate the extra file system identity information stored in the data integrity segment, since, by its nature, this information only has meaning to the file system and not the RAID-level scrub. Depending on system load, data scrubs are initiated on Sunday evenings. From our data, we find that an entire RAID group is scrubbed approximately once every two weeks on an average. However, we cannot ascertain from the data that every disk in the study has been scrubbed.

2.3 Corruption Classes

This study focuses on disk block corruptions caused by both hardware and software errors. Hardware bugs include bugs in the disk drive or the disk shelf firmware, bad memory, and adapter failures. Software bugs could also cause some corruption. In many cases, the cause of corruption cannot be identified. We detect different forms of corruption using the different data protection mechanisms in place. As mentioned earlier, we distinguish between these forms in our study. Table 1 gives a summary of these corruption classes.

- **Checksum mismatches (CMs):** This corruption class refers to cases where the corruption is detected from mismatched data and checksum. The cause could be (i) data content corrupted by components within the data path, or (ii) a torn write, wherein only a portion of the data block is written successfully, or (iii) a misdirected write, wherein the data is written to either the wrong disk or the wrong location on disk, thus overwriting and corrupting data [3, 15]. Checksum mismatches can be detected anytime a disk block is read (file system reads, data scrubs, RAID reconstruction and so on).

- **Identity discrepancies (IDs):** This corruption class refers to a mismatch detected when a disk block identity check is performed during a file system read. The cause could be (i) a lost write, which typically occurs because a write destined for disk is not written but thought of as written, or (ii) a misdirected write, where the original disk location is not updated. We are aware of actual cases when the disk firmware replied successfully to a write that was never written to stable media. Identity discrepancies can be detected only during file system reads.

- **Parity inconsistencies (PIs):** This corruption class refers to a mismatch between the parity computed from data blocks and the parity stored on disk despite the individual checksums being valid. This error could be caused by lost or misdirected writes, in-memory corruptions, processor miscalculations, and software bugs. Parity inconsistencies are detected only during data scrubs.

Our study primarily focuses on checksum mismatches, since we find that these corruptions occur much more frequently.

3 Methodology

This section describes some terminology, our data collection and analysis methodology, and notation used to discuss our results.

3.1 Terminology

We use the following terms in the remaining sections.

Disk class Enterprise Class or nearline disk drives with respectively Fibre Channel and ATA interfaces.

Disk family A particular disk drive product. The same product (and hence a disk family) may be offered in different capacities. Typically, disks in the same family only differ in the number of platters and/or read/write heads [17].

Disk model The combination of a disk family and a particular disk size. Note that this term does not imply an analytical or simulation model.

Disk age The amount of time a disk has been in the field since its ship date, rather than the manufacture date. In practice these two values are typically within a month of each other.

Corrupt block This term refers to a 4-KB file system block with a checksum mismatch.

Corrupt disk This term is used to refer to a disk drive that has at least one corrupt block.

3.2 Data Collection and Analysis

We now describe our data collection and analysis methodology and some limitations.

Data collection: The storage system has a built-in, low-overhead mechanism called Autosupport to log important system events back to a central repository. These messages can be enabled for a variety of system events including disk errors. Not all customers enable logging, although a large percentage do. Those that do, sometimes do so only after some period of initial use. These logs allow customized support based on observed events. Although these logs are primarily intended for support, they have also been utilized for analyzing various disk errors. In addition to our corruption study, this repository (the “Network Appliance Autosupport Database”) has been used in disk failure [11] and latent sector error [2] studies.

Analysis: We study corruption instances that were logged in tens of thousands of storage systems for a period of 41 months starting in January 2004. These systems belong to a range of different models, run different versions of storage-controller software (perhaps with one or more updates during the study period) and contain many different models or versions of hardware components. In order to have a complete history of the activities of the disks used in the study, we constrain our sample to only those disks that were shipped after January 2004. Our sample consists of 1.53 million disk drives. These drives belong to 14 disk families and 31 distinct models. To derive statistically significant results, we often further constrain the sample set depending on the analysis being performed. For example, we sometimes use shorter time

periods for our analysis so as to maximize the number of models we can study; clearly not all disk families and models have been in the field for the same duration. The disk models we consider for each study may have one of the following constraints:

- Model has at least 1000 disks in the field for time period being considered.
- Model has at least 1000 disks in the field and at least 15 corrupt disks for time being considered.

The first constraint is used for studies of factors that impact checksum mismatches, while other studies use the second constraint. In addition to the constraints on the model sample, we often restrict our data to include only the first 17 months since a drive was shipped. This helps make results more comparable, since many of the drives in the study were shipped on different dates and have been in the field for different amounts of time.

While we usually present data for individual disk models, we sometimes also report averages (mean values) for nearline disks and enterprise class disks. Since the sample size for different disk models per disk class varies considerably, we weigh the average by the sample size of each disk model in the respective class.

Limitations: The study has a few limitations that mostly stem from the data collection process. First, for a variety of reasons, disks may be removed from the system. Our study includes those disks up to the point of their removal from the system. Therefore, we may not observe errors from otherwise error prone disks after some period of time. Second, since the logging infrastructure has been built with customized support as the primary purpose, the data can be used to answer most but not all questions that are interesting for a study such as ours. For example, while we can identify the exact disk when an error is detected during a scrub, we cannot verify that every disk in the study has been scrubbed periodically in the absence of errors.

3.3 Notation

We denote each disk drive model as $\langle family-type \rangle$. For anonymization purposes, *family* is a single letter representing the disk family (e.g., Quantum Fireball EX) and *type* is a single number representing the disk's particular capacity. Although capacities are anonymized, relative sizes within a family are ordered by the number representing the capacity. For example, n-2 is larger than n-1, and n-3 is larger than both n-1 and n-2. The anonymized capacities do not allow comparisons across disk families. Disk families from A to E (upper case letters) are nearline disk families, while families from f to o (lower case letters) are enterprise class disk families. Lines on graphs labeled *NL* and *ES* represent the weighted average for nearline and enterprise class disk models respectively.

We present data as the probability of developing x checksum mismatches for a particular sample of disks. The notation $P(X_T \geq L)$ denotes the probability of a disk developing at least L checksum mismatches within T months since the disk's first use in the field.

4 Checksum Mismatches

This section presents the results of our analysis of checksum mismatches. We first provide basic statistics on the occurrence of checksum mismatches in the entire population of disk drives. We then examine various factors that affect the probability of developing checksum mismatches. Next, we analyze various characteristics of checksum mismatches, such as spatial locality. Further, we look for correlations between occurrence of checksum mismatches and other system or disk errors. Finally, we analyze the source of the disk requests that discovered the mismatches.

4.1 Summary Statistics

During the 41-month period covered by our data we observe a total of about 400,000 checksum mismatches. Of the total sample of 1.53 million disks, 3855 disks developed checksum mismatches – 3088 of the 358,000 nearline disks (0.86%) and 767 of the 1.17 million enterprise class disks (0.065%). Using our probability representation, $P(X_t \geq 1) = 0.0086$ for nearline disks, and $P(X_t \geq 1) = 0.00065$ for enterprise class disks without any restriction on time, t . This indicates that nearline disks may be more susceptible to corruption leading to checksum mismatches than enterprise class disks. On average, each disk developed 0.26 checksum mismatches. Considering only corrupt disks, that is disks that experienced at least one checksum mismatch, the mean number of mismatches per disk is 104, the median is 3 and the mode (i.e. the most frequently observed value) is 1 mismatch per disk. The maximum number of mismatches observed for any single drive is 33,000.

4.2 Factors

We examine the dependence of checksum mismatches on various factors: disk class, disk model, disk age, disk size, and workload.

4.2.1 Disk Class, Model and Age

Figures 2 and 3 show the probability of a disk developing checksum mismatches as it ages for nearline and enterprise class disks respectively. The graphs plot the cumulative distribution function of the time until the first

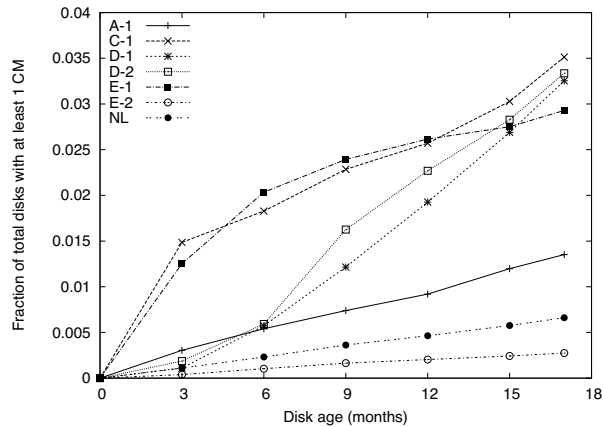


Figure 2: **Impact of disk age on nearline disks.** The probability that a disk develops checksum mismatches as it ages is shown for nearline disk models. Note that the probability is cumulative.

checksum mismatch occurs. The figures can be represented as $P(X_t \geq 1)$ for $t = \{3, 6, 9, 12, 15, 17\}$ months, i.e., the probability of at least one checksum mismatch after t months. Note the different Y-axis scale for the nearline and enterprise class disks.

We see from the figures that checksum mismatches depend on disk class, disk model and disk age.

Observation 1 Nearline disks (including the SATA/FC adapter) have an order of magnitude higher probability of developing checksum mismatches than enterprise class disks.

Figure 2 (line ‘NL’ – Nearline average) shows that 0.66% of nearline disks develop at least one mismatch during the first 17 months in the field ($P(X_{17} \geq 1) = 0.0066$), while Figure 3(b) (line ‘ES’) indicates that only 0.06% of enterprise class disks develop a mismatch during that time ($P(X_{17} \geq 1) = 0.0006$).

Observation 2 The probability of developing checksum mismatches varies significantly across different disk models within the same disk class.

We see in Figure 2 that there is an order of magnitude difference between models ‘C-1’ and ‘E-2’ for developing at least one checksum mismatch after 17 months; i.e., $P(X_{17} \geq 1) = 0.035$ for ‘C-1’ and 0.0027 for ‘E-2’.

Observation 3 Age affects different disk models differently with respect to the probability of developing checksum mismatches.

On average, as nearline disks age, the probability of developing a checksum mismatch is fairly constant, with some variation across the models. As enterprise class disks age, the probability of developing the first checksum mismatch decreases after about 6-9 months and then stabilizes.

4.2.2 Disk Size

Observation 4 There is no clear indication that disk size affects the probability of developing checksum mismatches.

Figure 4 presents the fraction of disks that develop checksum mismatches within 17 months of their ship-date (i.e., the rightmost data points from Figures 2 and 3; $P(X_{17} \geq 1)$). The disk models are grouped within their families in increasing size. Since the impact of disk size on the fraction of disks that develop checksum mismatches is not constant across all disk families (it occurs in only 7 out of 10 families), we conclude that disk size does not necessarily impact the probability of developing checksum mismatches.

4.2.3 Workload

Observation 5 There is no clear indication that workload affects the probability of developing checksum mismatches.

The systems in the study collect coarse workload data including the number of read and write operations, and the number of blocks read and written for each week of our study. To study the effect of workload on checksum mismatches, we computed the correlation coefficient between the workload data and the number of checksum mismatches observed in the system.

We find that in all cases the correlation coefficient is less than 0.1 (in fact, in most cases less than 0.001), indicating no significant correlation between workload and checksum mismatches. However, these results might be due to having only coarse per-system rather than per-drive workload data. A system consists of at least 14 disks and can have as many as several hundred disks. Aggregating data across a number of disks might blur existing correlations between an individual drive’s workload and corruption behavior.

4.3 Characteristics

In this subsection, we explore various characteristics of checksum mismatches. First, we analyze the number of mismatches developed by corrupt disks. Then, we examine whether mismatches are independent occurrences. Finally, we examine whether the mismatches have spatial or temporal locality.

4.3.1 Checksum mismatches per corrupt disk

Figure 5 shows the cumulative distribution function of the number of checksum mismatches observed per corrupt disk, i.e. the Y-axis shows the fraction of corrupt

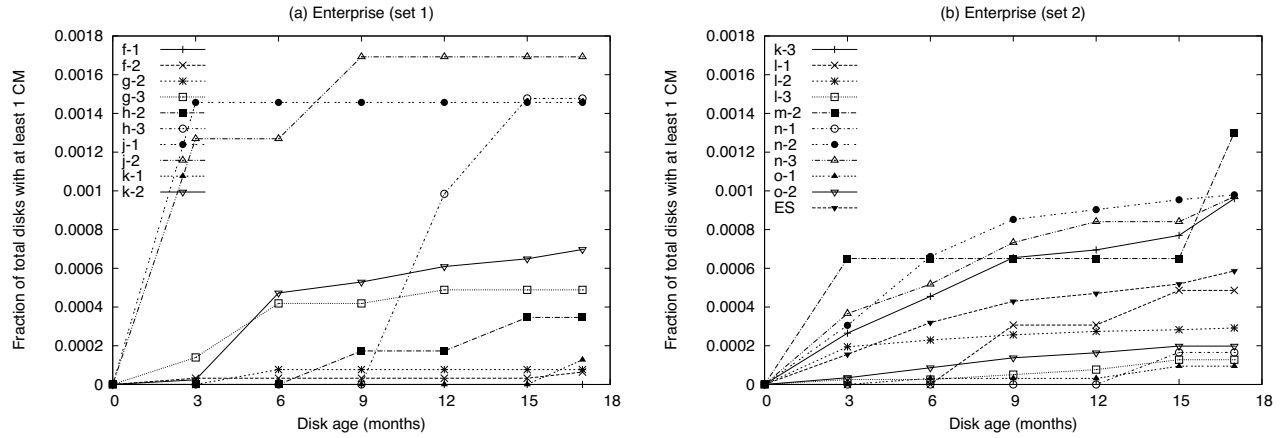


Figure 3: **Impact of disk age on enterprise class disks.** The probability that a disk develops checksum mismatches as it ages is shown for enterprise class disk models. Note that the probability is cumulative.

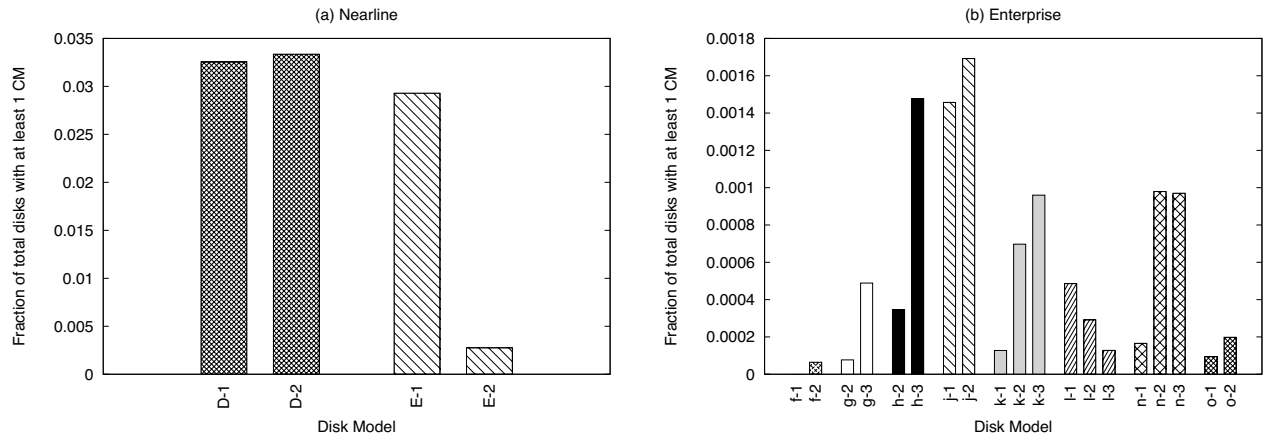


Figure 4: **The impact of disk size.** The figures show the fraction of disks with at least one checksum mismatch within 17 months of shipping to the field for (a) nearline disk models, and (b) enterprise class disk models.

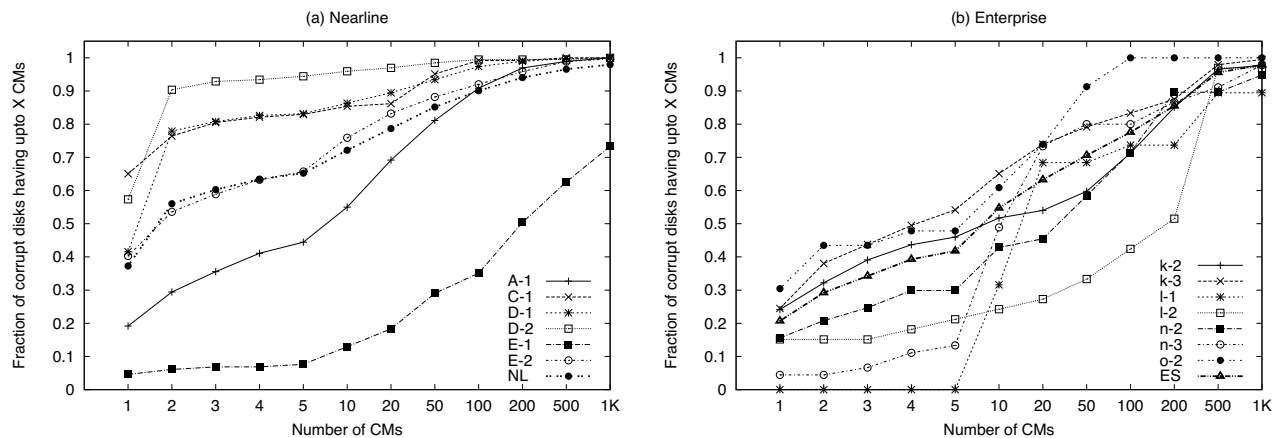


Figure 5: **Checksum mismatches per corrupt disk.** The fraction of corrupt disks as a function of the number of checksum mismatches that develop within 17 months after the ship date for (a) nearline disk models and (b) enterprise class disk models. Note that the x-axis is not linear in scale – the lines in the graph are used only to help distinguish the points of different disk models, and their slopes are not meaningful.

disks that have fewer than or equal to X number of corrupt blocks. The figure can be represented as $P(X_{17} \leq x | X_{17} \geq 1)$ for $x = \{1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500, 1000\}$.

Observation 6 *The number of checksum mismatches per corrupt disk varies greatly across disks. Most corrupt disks develop only a few mismatches each. However, a few disks develop a large number of mismatches.*

Figure 5 shows that a significant fraction of corrupt disks (more than a third of all corrupt nearline disks and more than a fifth of corrupt enterprise disks) develop only one checksum mismatch. On the other hand, a small fraction of disks develop several thousand checksum mismatches. The large variability in the number of mismatches per drive is also reflected in the great difference between the mean and median: while the median is only 3 mismatches per drive, the mean is 78.

A more detailed analysis reveals that the distributions exhibit heavy tails. A large fraction of the total number of checksum mismatches observed in our study is experienced by a very small fraction of the corrupt disks. More precisely, 1% of the corrupt disks (the top 1% corrupt disks with the largest number of mismatches) produce more than half of all mismatches recorded in the data.

Observation 7 *On average, corrupt enterprise class disks develop many more checksum mismatches than corrupt nearline disks.*

Figure 5(a) (line ‘NL’) and Figure 5(b) (line ‘ES’) show that within 17 months 50% of corrupt disks (i.e., the median) develop about 2 checksum mismatches for nearline disks, but almost 10 for enterprise class disks. The trend also extends to a higher percentage of corrupt disks. For example, 80% of nearline corrupt disks have fewer than 20 mismatches, whereas 80% of enterprise class disks have fewer than 100 mismatches. Given that very few enterprise class disks develop checksum mismatches in the first place, in the interest of reliability and availability, it might make sense to replace the enterprise class disk when the first mismatch is detected.

Observation 8 *Checksum mismatches within the same disk are not independent.*

We find that the conditional probability of developing further checksum mismatches, given that a disk has at least one mismatch, is higher than the probability of developing the first mismatch. For example, while the probability that a nearline disk will develop one or more checksum mismatches in 17 months is only 0.0066, the conditional probability of developing more than 1 mismatch given that the disk already has one mismatch is as high as 0.6 (1 minus 0.4, the probability of exactly 1 block developing a checksum mismatch in Figure 5).

Finally, it is interesting to note that nearline disk model ‘E-1’ is particularly aberrant – around 30% of its corrupt disks develop more than 1000 checksum mismatches. We are currently investigating this anomaly.

4.3.2 Dependence between disks in same system

Observation 9 *The probability of a disk developing a checksum mismatch is not independent of that of other disks in the same storage system.*

While most systems with checksum mismatches have only one corrupt disk, we do find a considerable number of instances where multiple disks develop checksum mismatches within the same storage system. In fact, one of the systems in the study that used nearline disks had 92 disks develop checksum mismatches. Taking the maximum number of disks in the systems in the study into consideration, the probability of 92 disks developing errors independently is less than $1e - 12$, much less than $1e - 05$, the approximate fraction of systems represented by one system. This dependence is perhaps indicative of a common corruption-causing component, such as a shelf controller or adapter. We are aware of such components causing corruptions.

4.3.3 Spatial Locality

We measure spatial locality by examining whether each corrupt block has another corrupt block (a *neighbor*) within progressively larger regions (*locality radius*) around it on the same disk. For example, if in a disk, blocks numbered 100, 200 and 500 have checksum mismatches, then blocks 100 and 200 have one neighbor at a locality radius of 100, and all blocks (100, 200, and 500) have at least one neighbor at a locality radius of 300.

Figure 6 shows the percentage of corrupt blocks that have at least one neighbor within different locality radii. Since a larger number of checksum mismatches will significantly skew the numbers, we consider only disks with 2 to 10 mismatches. The figure can be represented as $P(X_t^r \geq 1 | 2 \leq X_t \leq 10)$. X_t^r is the number of corrupt blocks in block numbers $< a - r, a + r >$ around corrupt block a (but excluding a itself). The values for radius r are $\{1, 10, 100, \dots, 100M\}$ blocks, and $0 < t \leq 41$ months. The figure also includes a line *Random* that signifies the line that would be obtained if the checksum mismatches were randomly distributed across the block address space. This line can be used as a comparison point against the other lines. Note that this line is at 0 for most of the graph, signifying that there is no spatial locality for a random distribution.

For the actual data for the different disk models, we see that most disk models are much higher on the graph than *Random* when the x-axis value is 1; for more than

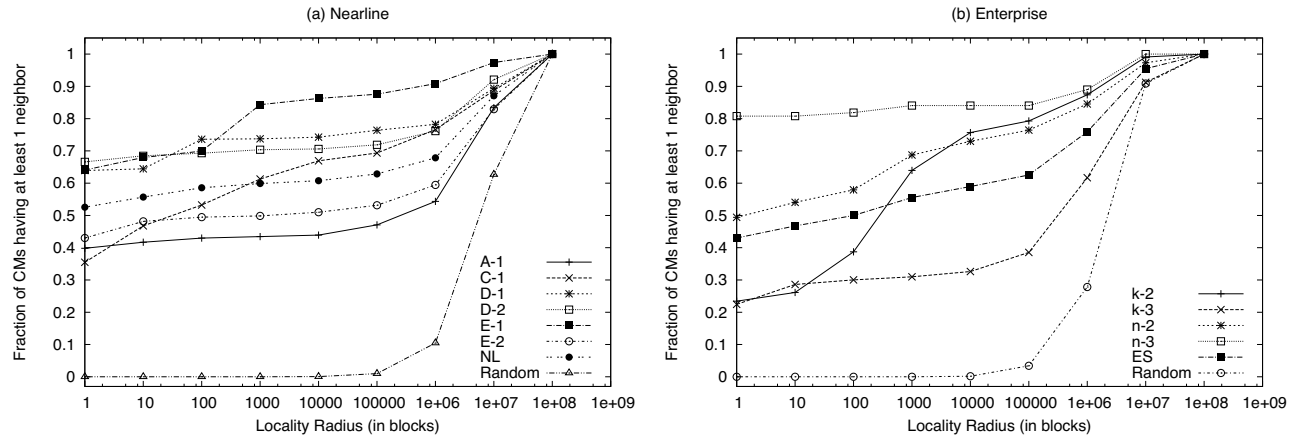


Figure 6: Spatial Locality. The graphs show the probability of another checksum mismatch within a given radius (disk block range) of one checksum mismatch. Each figure also includes a line labeled “Random” corresponding to when the same number of mismatches (as nearline and enterprise class respectively) are randomly distributed across the block address space. Only disks with between 2 and 10 mismatches are included.

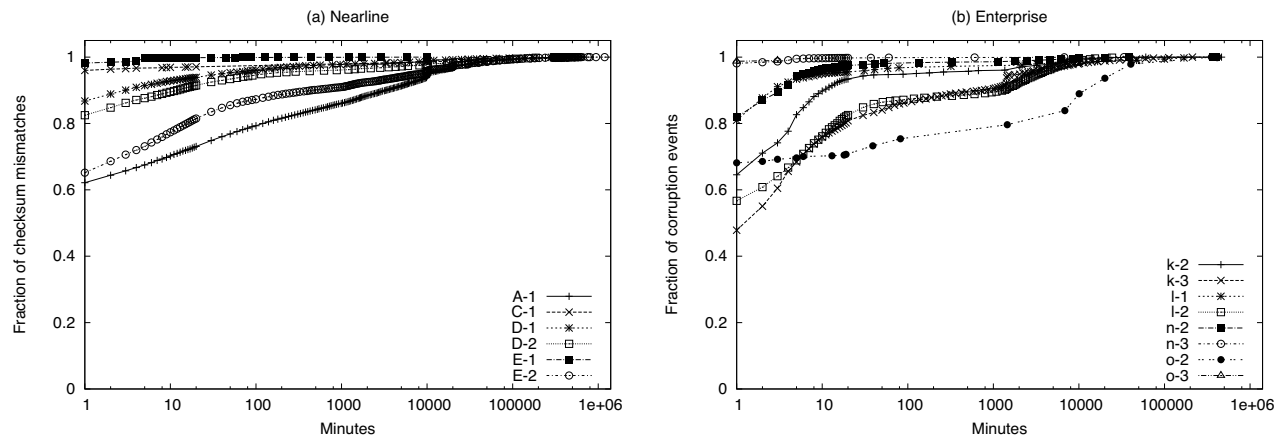


Figure 7: Inter-arrival times. The graphs show the cumulative distribution of the inter-arrival times of checksum mismatches per minute. The fraction of mismatches per model is plotted against time. The arrival times are binned by minute.

50% of the corrupt blocks in nearline disks and more than 40% of the corrupt blocks in enterprise class disks, the immediate neighboring block also has a checksum mismatch (on disks with between 2 and 10 mismatches). These percentages indicate very high spatial locality.

Observation 10 *Checksum mismatches have very high spatial locality. Much of the observed locality is due to consecutive disk blocks developing corruption. Beyond consecutive blocks, the mismatches show very little spatial locality.*

We see from the figures that, while the lines for the disk models start at a very high value when the x-axis value is 1, they are almost flat for most of the graph, moving steeply upwards to 1 only towards the end (x-axis values more than $1e+06$). This behavior shows that most of the spatial locality is due to consecutive blocks

developing checksum mismatches. However, it is important to note that even when the consecutive mismatch cases are disregarded, the distribution of the mismatches still has spatial locality.

Given the strong correlation between checksum mismatches in consecutive blocks, it is interesting to examine the run length of consecutive mismatches, i.e., how many consecutive blocks have mismatches. We find that, among drives with at least 2 checksum mismatches (and no upper bound on mismatches), on average 3.4 consecutive blocks are affected. In some cases, the length of consecutive runs can be much higher than the average. About 3% of drives with at least 2 mismatches see one or more runs of 100 consecutive blocks with mismatches. 0.7% of drives with at least 2 mismatches see one or more runs of 1000 consecutive mismatches.

4.3.4 Temporal Locality

Figure 7 shows the fraction of checksum mismatches that arrive (are detected) within x minutes of a previous mismatch. The figure can be represented as $P(X_{t+x} \geq k+1 | X_t = k \wedge X_T \geq k+1)$ for $k \geq 1, 0 \leq t < T \leq 41$ months, and $1 \leq x \leq 1e+06$ minutes.

Observation 11 *Most checksum mismatches are detected within one minute of a previous detection of a mismatch.*

The figure shows that the temporal locality for detecting checksum mismatches is extremely high. This behavior may be an artifact of the manner in which the detection takes place (by scrubbing) and the fact that many mismatches are spatially local and are therefore likely to be discovered together. Further analysis shows that this is not necessarily the case.

Observation 12 *Checksum mismatches exhibit temporal locality over larger time windows and beyond the effect of detection time as well.*

In order to remove the impact of detection time, we examine temporal locality over larger time windows. For each drive, we first determine the number of checksum mismatches experienced in each 2-week time window that the drive was in the field and then compute the autocorrelation function (ACF) on the resulting time series. The autocorrelation function (ACF) measures the correlation of a random variable with itself at different time lags l . The ACF can be used to determine whether the number of mismatches in one two-week period of our time-series is correlated with the number of mismatches observed l 2-week periods later. The autocorrelation coefficient can range between 1 (high positive correlation) and -1 (high negative correlation). A value of zero would indicate no correlation, supporting independence of checksum mismatches.

Figure 8 shows the resulting ACF. The graph presents the average ACF across all drives in the study that were in the field for at least 17 months and experienced checksum mismatches in at least two different 2-week windows. Since the results are nearly indistinguishable for nearline and enterprise class drives, individual results are not given. If checksum mismatches in different 2-week periods were independent (no temporal locality on bi-weekly and larger time-scales) the graph would be close to zero at all lags. Instead we observe strong autocorrelation even for large lags in the range of up to 10 months.

4.4 Correlations with other error types

Our system logs data on various other system and disk errors as well. We attempted to establish correlations for

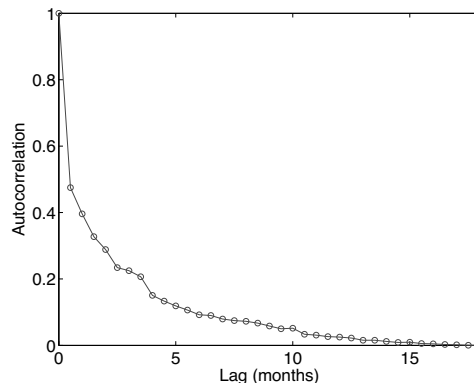


Figure 8: Temporal Autocorrelation. *The graph shows the autocorrelation function for the number of checksum mismatches per 2-week time windows. This representation of the data allows us to study temporal locality of mismatches at larger time-scales without being affected by the time of detection.*

checksum mismatches with system resets, latent sector errors, and not-ready-conditions.

Observation 13 *Checksum mismatches correlate with system resets.*

The conditional probability of a system reset at some point of time, given that one of the disks in the system has a checksum mismatch, is about 3.7 times the unconditional probability of a system reset. After a system reset, the system performs higher-level recovery operations; for example, a thorough file system integrity check may be run.

Observation 14 *There is a weak positive correlation between checksum mismatches and latent sector errors.*

The probability of a disk developing latent sector errors, $P(Y_t \geq 1)$, is 0.137 for nearline disks and 0.026 for enterprise class disks (Y is the number of latent sector errors, $0 < t \leq 41$ months). The conditional probability $P(Y_t \geq 1 | X_t \geq 1) = 0.195$ for nearline disks and 0.0556 for enterprise class disks. Thus, the conditional probability of a latent sector error, given that a disk has checksum mismatch, is about 1.4 times the unconditional probability of a latent sector error in the case of nearline disks and about 2.2 times the unconditional probability for enterprise class disks. These values indicate a weak positive correlation between the two disk errors.

In order to test the statistical significance of this correlation we performed a chi-square test for independence. We find that we can with high confidence reject the hypothesis that checksum mismatches and latent sector errors are independent, both in the case of nearline disks and enterprise class disks (confidence level of more than 99.999%). Interestingly, the results vary if we repeat the

chi-square test separately for each individual disk model (including only models that had at least 15 corrupt disks). We can reject independence with high certainty (at least 95% confidence) for only four out of seven nearline models (B-1, C-1, D-1, E-2) and two out of seven enterprise class models (l-1, n-3).

Observation 15 *There is a weak correlation between checksum mismatches and not-ready-conditions.*

The probability of a disk developing not-ready-conditions, $P(Z_t \geq 1)$, is 0.18 for nearline and 0.03 for enterprise class disks. $P(Z_t \geq 1|X_t \geq 1)$ is 0.304 for nearline and 0.0155 for enterprise class disks. Thus, the conditional probability of a not-ready-condition, given that a disk has checksum mismatch, is about 1.7 times the unconditional probability of a not-ready-condition in the case of nearline disks and about 0.5 times the unconditional probability for enterprise class disks. These values indicate mixed behavior – a weak positive correlation for nearline disks and a weak negative correlation for enterprise class disks.

In order to test the statistical significance of the correlation between not-ready-conditions and checksum mismatches, we again perform a chi-square test for independence. We find that for both nearline and enterprise disks we can reject the hypothesis that not-ready conditions and random corruptions are independent with more than 96% confidence. We repeat the same test separately for each disk model (including only models that had at least 15 corrupt disks). In the case of nearline disks, we can reject the independence hypothesis for all models, except for two (A-1 and B-1) at the 95% confidence level. However, in the case of enterprise class disks, we cannot reject the independence hypothesis for any of the individual models at a significant confidence level.

4.5 Discovery

Figure 9 shows the distribution of requests that detect checksum mismatches into different request types. There are five types of requests that discover checksum mismatches: (i) Reads by the file system (*FS Read*) (ii) Partial RAID stripe writes by the RAID layer (*Write*) (iii) Reads for disk copy operations (*Non-FS Read*) (iv) Reads for data scrubbing (*Scrub*), and (v) Reads performed during RAID reconstruction (*Reconstruction*).

Observation 16 *Data scrubbing discovers a large percentage of the checksum mismatches for many of the disk models.*

We see that on the average data scrubbing discovers about 49% of checksum mismatches in nearline disks (NL in the figure), and 73% of the checksum mismatches in enterprise class disks (ES in the figure). It is quite

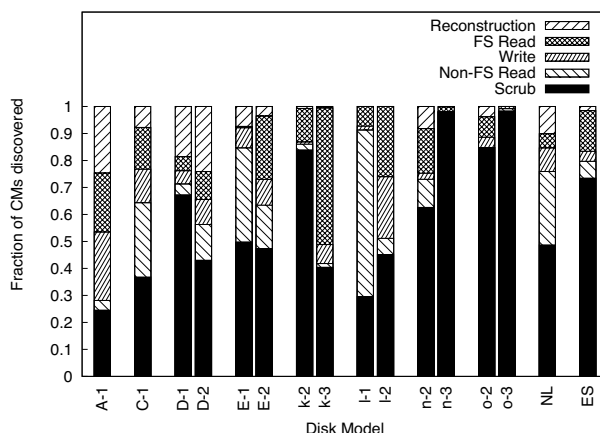


Figure 9: Request type analysis. *The distribution of requests that discover checksum mismatches across the request types scrub, non-file system read (say, disk copy), write (of partial RAID stripe), file system read, and RAID reconstruction.*

possible that these checksum mismatches may not have been discovered in the absence of data scrubbing, potentially exposing the system to double failures and data loss. We do not know the precise cause for the disparity in percentages between nearline and enterprise class disks; one possibility this data suggests is that systems with nearline disks perform many more disk copy operations (*Non-FS Read*), thus increasing the percentage for that request type.

Observation 17 *RAID reconstruction encounters a non-negligible number of checksum mismatches.*

Despite the use of data scrubbing, we find that RAID reconstruction discovers about 8% of the checksum mismatches in nearline disks. For some models more than 20% RAID reconstruction. This observation implies that (a) data scrubbing should be performed more aggressively, and (b) systems should consider protection against double disk failures [1, 4, 5, 9, 10, 12].

4.6 Comparison with Latent Sector Errors

In this subsection, we compare the characteristics of checksum mismatches, with the characteristics of latent sector errors, identified in a recent study [2].

Table 2 compares the behavior of checksum mismatches and latent sector errors. Some of the interesting similarities and differences are as follows.

Frequency: The probability of developing checksum mismatches is about an order of magnitude smaller than that for latent sector errors. However, given that customers use a few million disk drives, it is important to handle both kinds of errors. Also, since latent sector errors are more likely to be detected, it is more likely that an undetected checksum mismatch will lead to data loss.

Characteristic	Latent sector errors		Checksum mismatches	
	Nearline	Enterprise	Nearline	Enterprise
% disks affected per year (avg)	9.5%	1.4%	0.466%	0.042%
As disk age increases, P(1st error)	increases	remains constant	remains fairly constant	decreases, then stabilizes
As disk size increases, P(1st error)	increases	increases	unclear	unclear
No. of errors per disk with errors (80 percentile)	about 50	about 50	about 100	about 100
Are errors independent ?	No	No	No	No
Spatial locality	at 10 MB	at 10 MB	at 4KB	at 4KB
Temporal locality	very high	very high	very high	very high
Known Correlations	not-ready-conditions	recovered errors	system resets, not-ready-conditions	system resets, not-ready-conditions

Table 2: Checksum mismatches vs. Latent sector errors. *This table compares our findings on checksum mismatches with characteristics of latent sector errors identified by a recent study, for both nearline and enterprise class disk models. In addition to listed correlations, latent sector errors and checksum mismatches share a weak positive correlation with each other.*

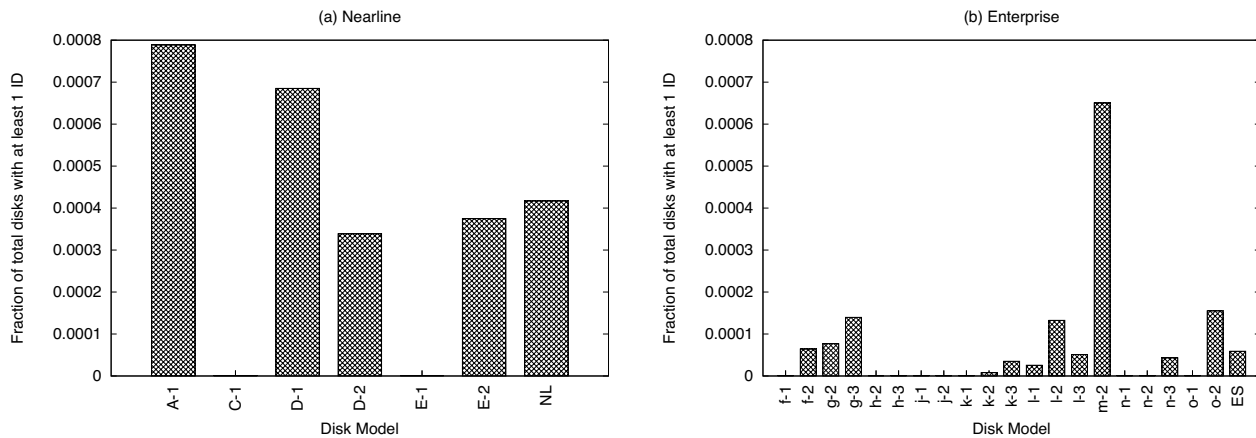


Figure 10: Identity Discrepancies. *The figures show the fraction of disks with at least one identity discrepancy within 17 months of shipping to the field for (a) nearline disk models, and (b) enterprise class disk models.*

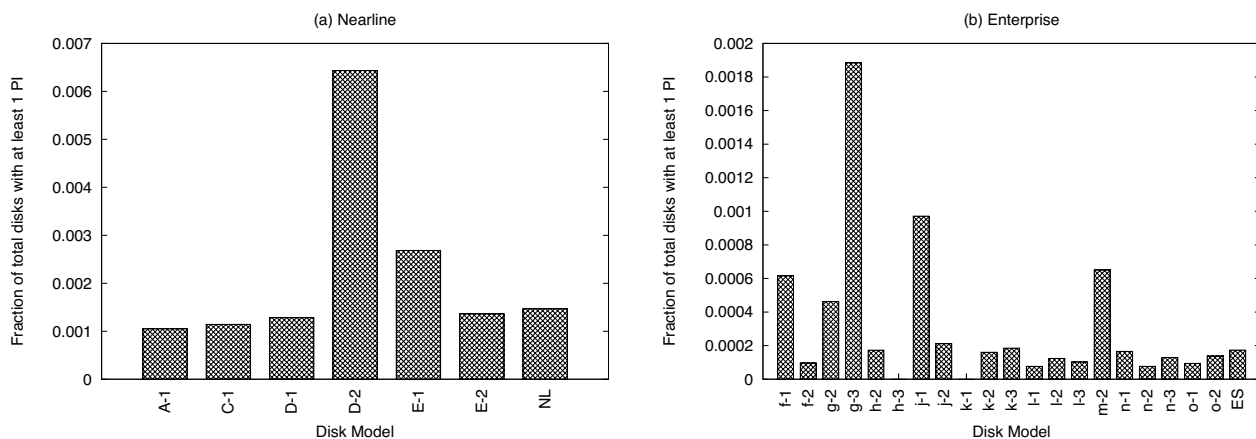


Figure 11: Parity Inconsistencies. *The figures show the fraction of disks with at least one parity inconsistency within 17 months of shipping to the field for (a) nearline disk models, and (b) enterprise class disk models.*

Disk model: The nearline disk model E-2 has the highest percentage of disks developing latent sector errors, but the lowest percentage of disks developing checksum mismatches within the set of nearline disk models.

Impact of disk class: For both latent sector errors and checksum mismatches, enterprise class disks are less likely to develop an error than nearline disks. Surprisingly, however, in both cases, enterprise class disks develop a higher number of errors than nearline disks, once an error has developed.

Spatial locality: Both latent sector errors and checksum mismatches show high spatial locality. Interestingly, the difference in the locality radii that capture a large fraction of errors – about 10 MB for latent sector errors versus consecutive blocks for checksum mismatches – provides an insight into how the two errors could be caused very differently. Latent sector errors may be caused by media scratches that could go across tracks as opposed to consecutive sectors (hence a larger locality radius) while consecutive blocks may have checksum mismatches simply because the corruption(s) occurred when they were written together or around the same time.

5 Other Data Corruptions

This section presents our results on the frequency of occurrence for two corruption classes: identity discrepancies, and parity inconsistencies. These corruption classes are described in Section 2.3.

5.1 Identity Discrepancies

These errors were detected in a total 365 disks out of the 1.53 million disks. Figure 10 presents the fraction of disks of each disk model that developed identity discrepancies in 17 months. We see that the fraction is more than an order of magnitude lower than that for checksum mismatches for both nearline and enterprise class disks.

Since the fraction of disks that develop identity discrepancies is very low, the system recommends replacement of the disk once the first identity discrepancy is detected. It is important to note, that even though the number of identity discrepancies are small, silent data corruption would have occurred if not for the validation of the stored contextual file system information.

5.2 Parity Inconsistencies

These errors are detected by data scrubbing. In the absence of a second parity disk, one cannot identify which disk is at fault. Therefore, in order to prevent potential data loss on disk failure, the system fixes the inconsistency by rewriting parity. This scenario provides further motivation for double-parity protection schemes.

Figure 11 presents the fraction of disks of each disk model that caused parity inconsistencies within 17 months since ship date. The fraction is 4.4 times lower than that for checksum mismatches in the case of nearline disks and about 3.5 times lower than that for checksum mismatches for enterprise class disks.

These results assume that the parity disk is at fault. We believe that counting the number of incorrect parity disks reflect the actual number of error disks since: (i) entire shelves of disks are typically of the same age and same model, (ii) the incidence of these inconsistencies is quite low; hence, it is unlikely that multiple different disks in the same RAID group would be at fault.

6 Experience

This section uses results from our analysis of corruption and leverages our experience in protecting against data corruption to develop insights into how storage systems can be designed to deal with corruption. First, we describe unexplained corruption phenomena and anecdotal insight into the causes of corruption, and then we discuss the lessons learned from our experience. Finally, we list some questions that could be looked at in future data analysis.

6.1 Anecdotes

6.1.1 Some block numbers are worse

From analysis, we find that specific block numbers could be much more likely to experience corruption than other block numbers. This behavior was observed for the disk model ‘E-1’. Figure 12 presents for each block number, the number of disk drives of disk model ‘E-1’ that developed a checksum mismatch at that block number. We see in the figure that many disks develop corruption for a specific set of block numbers. We also verified that (i) other disk models did not develop multiple checksum mismatches for the same set of block numbers (ii) the disks that developed mismatches at the same block numbers belong to different storage systems, and (iii) our software stack has no specific data structure that is placed at the block numbers of interest.

These observations indicate that hardware or firmware bugs that affect specific sets of block numbers might exist. Therefore, RAID system designers may be well-advised to use *staggered* stripes such that the blocks that form a stripe (providing the required redundancy) are placed at different block numbers on different disks.

We also observed a large number of block-specific errors on other drive models. In at least one of these instances, the block contained a heavily read and written file system metadata structure – a structure akin to the

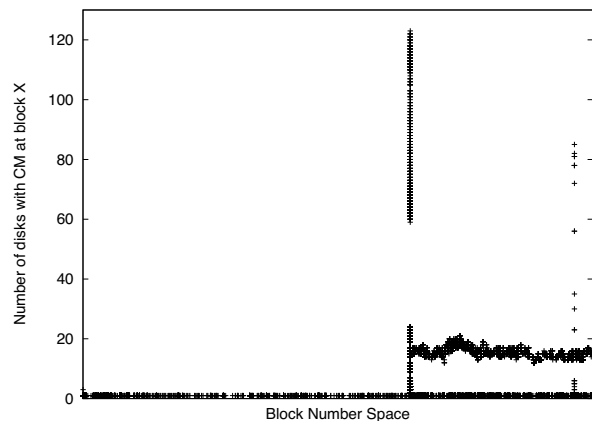


Figure 12: **Distribution of errors across block numbers.** For each disk block number, the number of disks of disk model E-1 that develop checksum mismatches at that block number is shown. The units on the x-axis have been omitted in order to anonymize the disk size of disk model E-1.

superblock. This suggests the importance of replicating important metadata structures [15, 20].

6.1.2 Other component failures

We have observed instances of the SATA/FC adapter causing data corruption in the case of disk models A-1, D-1 and D-2. Thus, it is very likely that the numbers for these disk models in Section 4 are influenced by faulty shelf controllers. Such behavior may also point to why different disks in the same system may not have independent failure behavior with respect to data corruption (Observation 9).

6.1.3 Cache flush bug

One of the disk drive models in the study had a bug specific to flushing the disk’s write cache. Upon reception of a cache flush command, the disk drive sometimes returned success without committing the data to stable storage on the disk medium. If, for any reason, the disk was then power-cycled the data just written was lost. However, thanks to block identity protection (and RAID), the storage system did not lose data.

6.2 Lessons Learned

We present some of the lessons learned from the analysis for corruption-proof storage system design. Some of these lessons are specific to RAIDs, while others can be applied to file systems as well.

- Albeit not as common as latent sector errors, data corruption does happen; we observed more than 400,000 cases of checksum mismatches. For some drive models

as many as 4% of drives develop checksum mismatches during the 17 months examined. Similarly, even though they are rare, identity discrepancies and parity inconsistencies do occur. Protection offered by checksums and block identity information is therefore well-worth the extra space needed to store them.

- A significant number (8% on average) of corruptions are detected during RAID reconstruction, creating the possibility of data loss. In this case, protection against double disk failures [1, 4, 5, 9, 10, 12] is necessary to prevent data loss. More aggressive scrubbing can speed the detection of errors, reducing the likelihood of an error during a reconstruction.

- Although, the probability of developing a corruption is lower for enterprise class drives, once they develop a corruption, many more are likely to follow. Therefore, replacing an enterprise class drive on the first detection of a corruption might make sense (drive replacement cost may not be a huge factor since the probability of first corruption is low).

- Some block numbers are much more likely to be affected by corruption than others, potentially due to hardware or firmware bugs that affect specific sets of block numbers. RAID system designers might be well advised to use *staggered* stripes such that the blocks that form the stripe are not stored at the same or nearby block number.

- Strong spatial locality suggests that redundant data structures should be stored distant from each other.

- The high degree of spatial and temporal locality also begs the question of whether many corruptions occur at the exact same time, perhaps when all blocks are written as part of the same disk request. This hypothesis suggests that important or redundant data structures that are used for recovering data on corruption should be written as part of different write requests spaced over time.

- Strong spatial and temporal locality (over long time periods) also suggests that it might be worth investigating how the locality can be leveraged for smarter scrubbing, e.g. trigger a scrub before it’s next scheduled time, when probability of corruption is high or *selective* scrubbing of an area of the drive that’s likely to be affected.

- Failure prediction algorithms in systems should take into account the correlation of corruption with other errors such as latent sector errors, increasing the probability of one error when an instance of the other is found.

6.3 Future Work

Future data analysis studies could focus on questions on data corruption and its causes that our current study does not answer. We discuss some such questions below.

- Our study looks at corruption numbers across different disk models. We find that the numbers vary significantly across disk models, suggesting that disks (and

their adapters) may directly or indirectly cause corruption most of the time. However, disks are only one of the storage stack components that could potentially cause corruption. A recent study shows that other storage subsystem components do have a significant impact on storage failures [11]). A future study could focus on corruption numbers across different models or versions of all hardware and software components. Such a study may also help pinpoint the exact sources of data corruption.

(ii) The impact of workload on the prevalence of data corruption is unclear, especially due to the lack of fine-grained disk-level workload information. Future studies may focus on obtaining this information along with recording disk corruption occurrences.

7 Related Work

There are very few studies of disk errors. Most disk fault studies examine either drive failures [14, 16, 17, 18] or latent sector errors [2]. Of the large scale drive failure studies, Schroeder and Gibson [16] analyze data from about 100,000 disks over a five year time period. They find that failure rate increases over time, and error rates are not constant with disk age. Pinheiro et al. [14] analyze data associated with over 100,000 disks over a nine month period. They use this data to analyze the correlation of disk failures to environmental factors and usage patterns. They find that the annualized failure rate is significantly higher for drives after the first year. Jiang et al. [11] study various aspects of storage subsystem failures. For example, they determine that subsystem components other than disks cause a significant fraction of observed failures. Shah and Elerath [7, 17, 18] have performed a number of studies on the reliability of disk drives. They find from these studies that there are many factors, including disk drive vintage, which influence the failure rate of disks and that there is a large variation between disk models and families.

There is a large body of work regarding techniques for detecting and recovering from data corruption. Sivathanu et. al [19] survey data integrity techniques within storage systems, and classify integrity violation types and detection and correction schemes. Prabhakaran et al. [15] develop a taxonomy to classify file system failures. They find that many of the file systems tested do not detect nor recover from most disk corruption errors. Many of the file systems tested use some form of disk block type checking (e.g., a magic-number for metadata), however lost or misdirected writes still cause corruption if the block's new type matched its previous type.

Using checksums to protect data integrity is an old concept, especially in communication systems. The Tandem NonStop Server [3] was designed to use end-to-end checksums at the host, writing it to disk with the data

and verifying it on read. However, the majority of file systems today still rely on disk drives to report internal errors. Two of the more recent file systems to use checksums to detect corrupted data include Sun's ZFS [21] and Google's GFS [8]. ZFS uses 64-bit checksums to validate the data path. A checksum is stored with every block pointer and is validated by the file system layer. If a checksum error is encountered and a mirrored copy is available, the mirrored copy is used to correct the data. GFS also uses checksums to protect data integrity. GFS distributes data across chunk servers that break the chunks into 64KB data blocks, each protected by a 32-bit checksum. On a checksum mismatch, the correct data is retrieved from a replica. Both file systems use some form of periodic scrubbing to validate data.

8 Conclusion

We have analyzed data corruption instances detected in 1.53 million disks used in our production storage systems. We classified these instances into three classes: checksum mismatches, identity discrepancies, and parity inconsistencies. We find that only a small fraction of disks develop checksum mismatches. An even smaller fraction are due to identity discrepancies or parity inconsistencies. Even with the small number of errors observed it is still critical to detect and recover from these errors since data loss is rarely tolerated in enterprise-class and archival storage systems.

We have identified various characteristics of checksum mismatches, including (i) the probability of developing the first checksum mismatch is almost an order of magnitude higher for nearline disks than for enterprise class disks, (ii) checksum mismatches are not independent and the number of mismatches per disk follows a heavy-tailed distribution, and (iii) checksum mismatches also show high spatial and temporal locality, encouraging system designers to develop schemes that spread redundant data with respect to both the on-disk location and time at which they are written.

We have obtained insights for corruption-proof storage design from the statistical and anecdotal evidence we have collected. We believe that such insights are essential for designing highly reliable storage systems.

Acknowledgments

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance.

We thank the Autosupport team at NetApp for their help in gathering data. Members of the RAID group

including Atul Goel, Tomislav Grcanac, Rajesh Sundaram, Jim Taylor, and Tom Theaker provided insightful comments on the analysis. David Ford, Stephen Harpster, Steve Kleiman, Brian Pawlowski and members of the Advanced Technology Group provided excellent input on the work. Ramon del Rosario, Jon Elerath, Tim Emami, Aziz Htite, Hung Lu, and Sandeep Shah helped understand characteristics of disks and systems in the study, and mine Autosupport data. Finally, we would like to thank our shepherd Alma Riska and the anonymous reviewers for their detailed comments that helped improve the paper.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

- [1] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, pages 62–72, Denver, Colorado, June 1997.
- [2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'07)*, San Diego, California, June 2007.
- [3] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, Jan. 2004.
- [4] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*, pages 245–254, Chicago, Illinois, Apr. 1994.
- [5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, California, Apr. 2004.
- [6] M. H. Darden. Data Integrity: The Dell—EMC Distinction. http://www.dell.com/content/topics/global.aspx/power/en/ps2q02_darden?c=us&cs=555&l=en&s=biz, May 2002.
- [7] J. G. Elerath and S. Shah. Server Class Disk Drives: How Reliable Are They. In *The Proceedings of the 50th Annual Reliability and Maintainability Symposium*, pages 151–156, Los Angeles, California, Jan. 2004.
- [8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing (Lake George), New York, October 2003.
- [9] J. L. Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, Dec. 2005.
- [10] J. L. Hafner, V. W. Deenadhayalan, K. Rao, and J. A. Tomlin. Matrix Methods for Lost Data Reconstruction in Erasure Codes. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, Dec. 2005.
- [11] W. Jiang, C. Hu, A. Kanevsky, and Y. Zhou. Is Disk the Dominant Contributor for Storage Subsystem Failures? A Comprehensive Study of Failure Characteristics. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, Feb. 2008.
- [12] C.-I. Park. Efficient Placement of Parity and Data to Tolerate Two Disk Failures in Disk Array Systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(11):1177–1184, Nov. 1995.
- [13] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.
- [14] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, Feb. 2007.
- [15] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, Oct. 2005.
- [16] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, Feb. 2007.
- [17] S. Shah and J. G. Elerath. Disk Drive Vintage and its Effect on Reliability. In *The Proceedings of the 50th Annual Reliability and Maintainability Symposium*, pages 163–167, Los Angeles, California, Jan. 2004.
- [18] S. Shah and J. G. Elerath. Reliability Analyses of Disk Drive Failure Mechanisms. In *The Proceedings of the 51st Annual Reliability and Maintainability Symposium*, pages 226–231, Alexandria, Virginia, Jan. 2005.
- [19] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the ACM Workshop on Storage Security and Survivability (StorageSS '05)*, pages 26–36, Fairfax, Virginia, November 2005.
- [20] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, Apr. 2004.
- [21] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.
- [22] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/mat/sample/0206tot_resiliency.html, Feb. 2006.
- [23] Information Technology: SCSI Primary Commands (SPC-2). Technical Report T10 Project 1236-D Revision 5, Sept. 1998.

BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage

Hyojun Kim and Seongjun Ahn

Software Laboratory of Samsung Electronics, Korea
{zartoven, seongjun.ahn}@samsung.com

Abstract

Flash memory has become the most important storage media in mobile devices, and is beginning to replace hard disks in desktop systems. However, its relatively poor random write performance may cause problems in the desktop environment, which has much more complicated requirements than mobile devices. While a RAM buffer has been quite successful in hard disks to mask the low efficiency of random writes, managing such a buffer to fully exploit the characteristics of flash storage has still not been resolved. In this paper, we propose a new write buffer management scheme called Block Padding Least Recently Used, which significantly improves the random write performance of flash storage. We evaluate the scheme using trace-driven simulations and experiments with a prototype implementation. It shows about 44% enhanced performance for the workload of MS Office 2003 installation.

1 Introduction

Flash memory has many attractive features such as low power consumption, small size, light weight, and shock resistance [3]. Because of these features, flash memory is widely used in portable storage devices and handheld devices. Recently, flash memory has been adopted by personal computers and servers in the form of onboard cache and solid-state disk (SSD).

Flash memory-based SSDs exhibit much better performance for random reads compared to hard disks because NAND flash memory does not have a seek delay. In a hard disk, the seek delay can be up to several milliseconds. For sequential read and write requests, an SSD has a similar or better performance than a hard disk [4]. However, SSDs exhibit worse performance for random writes due to the unique physical characteristics of NAND flash memory. The memory must be erased before it can be written. The unit of erase operation is

relatively large, typically a block composed of multiple pages, where a page is the access unit. To mask this mismatch between write and erase operations, SSDs use additional software, called the flash translation layer (FTL) [6, 14] whose function is to map the storage interface logical blocks to physical pages within the device. The SSD random write performance is highly dependent on the effectiveness of the FTL algorithm.

Different types of FTL algorithms exist. Mobile phones use relatively complicated algorithms, but simpler methods are used for flash memory cards and USB mass storage disks. In SSDs, the controller has restricted computing power and working random access memory (RAM) to manage a large quantity of NAND flash memory, up to tens of gigabytes. Therefore, the SSD FTL must attain the cost efficiency of flash memory cards rather than mobile phones.

To obtain better performance with restricted resources, some FTLs exploit locality in write requests. A small portion of the flash memory is set aside for use as a write buffer to compensate for the physical characteristics of NAND flash memory. With high access locality, the small write buffer can be effective. However, FTLs show poor performance for random writes with no locality. The poor performance of SSDs for random writes can significantly impact desktop and server systems, which may have more complicated write patterns; in these systems, multiple threads commonly request I/O jobs concurrently, resulting in complex write access patterns. Therefore, the random write performance is of increasing importance in SSD design.

Several different approaches exist to enhancing random write performance. We selected a method of using a RAM buffer inside the SSD because it is very realistic and can be easily applied to current SSD products regardless of their FTL algorithms. The issue, however, is how to use the RAM buffer properly. In the case of a hard disk, the elevator algorithm is used to minimize the head movements.

In this paper, we present a new write buffer management scheme called Block Padding Least Recently Used (BPLRU) to enhance the random write performance of flash storage. BPLRU considers the common FTL characteristics and attempts to establish a desirable write pattern with RAM buffering. More specifically, BPLRU uses three key techniques, block-level LRU, page padding, and LRU compensation. Block-level LRU updates the LRU list considering the size of the erasable block to minimize the number of merge operations in the FTL. Page padding changes the fragmented write patterns to sequential ones to reduce the buffer flushing cost. LRU compensation adjusts the LRU list to use RAM for random writes more effectively. Using write traces from several typical tasks and three different file systems, we show that BPLRU is much more effective than previously proposed schemes, both in simulation and in experiments with a real prototype.

The rest of the paper is organized as follows. Section 2 presents the background and related work. Section 3 explains the proposed buffer cache management schemes. Section 4 evaluates the BPLRU scheme, and Section 5 contains our conclusions.

2 Background and Related Work

2.1 Flash Memory

There are two types of flash memories: NOR and NAND flash memories [18]. NOR flash memory was developed to replace programmable read-only memory (PROM) and erasable PROM (EPROM), which were used for code storage; so it was designed for efficient random access. It has separate address and data buses like EPROM and static random access memory (SRAM). NAND-type flash memory was developed more recently for data storage, and so was designed to have a denser architecture and a simpler interface than NOR flash memory. NAND flash memory is widely used.

Flash memories have a common physical restriction; they must be erased before writing. In flash memory, the existence of an electric charge represents 1 or 0, and the charges can be moved to or from a transistor by an erase or write operation. Generally, the erase operation, which makes a storage cell represent 1, takes longer than the write operation, so its operation unit was designed to be bigger than the write operation for better performance. Thus, flash memory can be written or read a single page at a time, but it can be erased only block by block. A block consists of a certain number of pages. The size of a page ranges from a word to 4 KB depending on the type of device. In NAND flash memory, a page is similar to a hard disk sector and is usually 2 KB. For NOR type flash memory, the size of a page is just one word.

Flash memory also suffers from a limitation in the number of erase operations possible for each block. The insulation layer that prevents electric charges from dispersing may be damaged after a certain number of erase operations. In single level cell (SLC) NAND flash memory, the expected number of erasures per a block is 100,000 and this is reduced to 10,000 in multilevel cell (MLC) NAND flash memory. If some blocks that contain critical information are worn out, the whole memory becomes useless even though many serviceable blocks still exist. Therefore, many flash memory-based devices use wear-leveling techniques to ensure that blocks wear out evenly.

2.2 Flash Translation Layer

The FTL overcomes the physical restriction of flash memory by remapping the logical blocks exported by a storage interface to physical locations within individual pages [6]. It emulates a hard disk, and provides logical sector updates¹(Figure 1). The early FTLs used a log-structured architecture [23] in which logical sectors were appended to the end of a large log, and obsolete sectors were removed through a garbage collection process. This architecture is well suited for flash memory because it cannot be overwritten. We call this method *page mapping FTL* because a logical sector (or page) can be written to any physical page [14]. The FTL writes the requested sector to a suitable empty page, and it maintains the mapping information between the logical sector and the physical page separately in both flash and main memories because the information is necessary to read the sector later.

Page-mapping FTL sufficed for small flash memory sizes; its mapping information size was also small. However, with the development of NAND flash memory and its exponential size increase, the page-mapping method became ineffective. It requires a great deal of memory for its mapping information. In some cases, the mapping information must be reconstructed by scanning the whole flash memory at start-up, and this may result in long mount time. Therefore, a new memory-efficient algorithm was required for very large NAND flash memories.

The *block mapping FTL* which is used for the Smart Media card [26], is not particularly efficient because a sector update may require a whole block update. An improvement on this scheme, called the *hybrid mapping FTL* [15], manages block-level mapping like *block map-*

¹Even though the term *sector* represents physical block of data on a hard disk, it is commonly used as an access unit for the FTL because it emulates a hard disk. The size of logical sector in the FTL may be 512 B, 2 KB, or 4 KB for efficiency. We adopt the same convention in this paper.

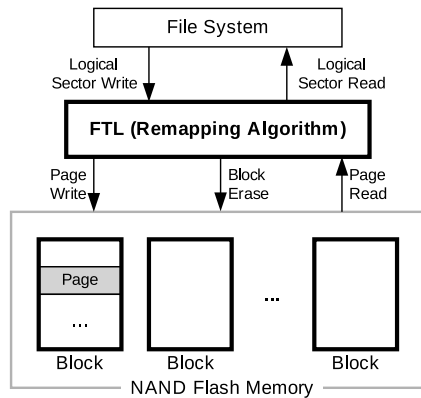


Figure 1: FTL and NAND flash memory. FTL emulates sector read / write functionalities of a hard disk to use conventional disk file systems on NAND flash memory.

ping FTL, but the sector position is not fixed inside the block. While this requires additional offset-level mapping information, the memory requirement is much less than in page mapping FTL.

Other FTLs were designed to exploit the locality of the write requests. If write requests are concentrated in a certain address range, some reserved blocks not mapped to any externally visible logical sectors can be used temporarily for those frequently updated logical sectors. When we consider the usage pattern of the flash storage, this is quite reasonable. Because the number of reserved blocks is limited, more flexible and efficient mapping algorithms can be applied to the reserved blocks while most data blocks use simple block mapping.

The *replacement block algorithm* [2] assigns multiple physical blocks to one logical block. It only requires block-level mapping information, which represents a physical block mapped to a particular logical block, and its creation order when multiple physical blocks exist. The *log-block FTL algorithm* [17] combines a coarse-grained mapping policy of a block mapping method with a fine-grained mapping policy of page mapping inside a block. Compared to the replacement block algorithm, it requires more mapping tables for log blocks, but can use reserved blocks more effectively. The log-block FTL algorithm is one of the most popular algorithms today because it combines competitive performance with rather low cost in terms of RAM usage and CPU power.

2.3 Log-block FTL

In the log-block FTL algorithm, sectors are always written to log blocks that use a fine-grained mapping policy allowing a sector to be in any position in a block. This is very efficient for concentrated updates. For example,

if Sector 0 is repeatedly written four times when a block consists of four pages, all pages in the log block can be used only for sector 0. When a log block becomes full, it merges with the old data block to make a new data block. The valid sectors in the log block and in the old data block are copied to a free block, and the free block becomes the new data block. Then, the log block and the old data block become free blocks.

Sometimes, a log block can just replace the old data block. If sectors are written to a log block from its first sector to the last sector sequentially, the log block gains the same status as the data block. In this case, the log block will simply replace the old data block, and the old data block will become a free block. This replacement is called the *switch merge* in log-block FTL.

Switch merge provides the ideal performance for NAND flash memory. It requires single-block erasure and N page writes, where N is the number of pages per block. To distinguish it from a *switch merge*, we refer to a normal merge as a *full merge* in this paper. The *full merge* requires two block erasures, N page reads, and N page writes.

2.4 Flash Aware Caches

Clean first LRU (CFLRU) [21, 22] is a buffer cache management algorithm for flash storage. It was proposed to exploit the asymmetric performance of flash memory read and write operations. It attempts to choose a clean page as a victim rather than dirty pages because writing cost is much more expensive. CFLRU was found to be able to reduce the average replacement cost by 26% in the buffer cache compared to the LRU algorithm. CFLRU is important because it reduces the number of writes by trading off the number of reads. However, this is irrelevant when only write requests are involved. Thus, it is not useful for enhancing random write performance.

The *flash aware buffer policy* (FAB) [10] is another buffer cache management policy used for flash memory. In FAB, the buffers that belong to the same erasable block of flash memory are grouped together, and the groups are maintained in LRU order: a group is moved to the beginning of the list when a buffer in the group is read or updated, or a new buffer is added to the group. When all buffers are full, a group that has the largest number of buffers is selected as victim. If more than one group has the same largest number of buffers, the least recently used of them is selected as a victim. All the dirty buffers in the victim group are flushed, and all the clean buffers in it are discarded. The main use of FAB is in portable media player applications in which the majority of write requests are sequential. FAB is very effective compared to LRU. Note that BPLRU targets random write patterns.

Jiang et al. proposed DULO [8], an effective buffer

cache management scheme to exploit both temporal and spatial locality. Even though it is not designed for flash storage, the basic concept of DULO is very similar to our proposed method. In DULO, the characteristics of a hard disk are exploited so that sequential access is more efficient than random access. Similarly, flash storage shows optimized write performance for sequential writes in a block boundary because most FTLs use the spatial locality of the block level. Therefore, we use a RAM buffer to create the desirable write pattern while the existing buffer caches are used to try to reduce the number of write requests.

Recently developed SSDs for desktop or server applications face quite different access patterns compared to mobile applications. In server systems, multiple processes may request disk access concurrently, which can be regarded as essentially as a random pattern because many processes exist.

An SSD has very different performance characteristics compared to a hard disk. For random read requests, the SSD is much better than a hard disk because it does not have any mechanical parts. This is one of the most important advantages of SSDs, offset, however, by the poor random write performance. Due to the characteristics of NAND flash memory and the FTL algorithm inside an SSD, the performance is now much worse than a hard disk and the life span of the SSD can be reduced by random write patterns.

Two different approaches are possible. First, the FTL may use a more flexible mapping algorithm such as page mapping, hybrid mapping, or the super-block FTL algorithm [13]. However, flexible mapping algorithms generally require more RAM, CPU power, and a long start-up time, so they may not be feasible for SSDs. The second approach involves embedding the RAM buffer in the SSDs just like in a hard disk. We propose a method for proper management of this write buffer.

3 BPLRU

We devised BPLRU as a buffer management scheme to be applied to the write buffer inside SSDs. BPLRU allocates and manages buffer memory only for write requests. For reads, it simply redirects the requests to the FTL. We chose to use all of available RAM inside an SSD as write buffers because most desktop and server computers have a much larger host cache that can absorb repeated read requests to the same block more effectively than the limited RAM on the SSD device.

Figure 2 shows the general system configuration considered in this paper. The host includes a CPU, a file system, and a device-level buffer cache on the host side. The SSD device includes the RAM for buffering writes, the FTL, and the flash memory itself. For the host buffer

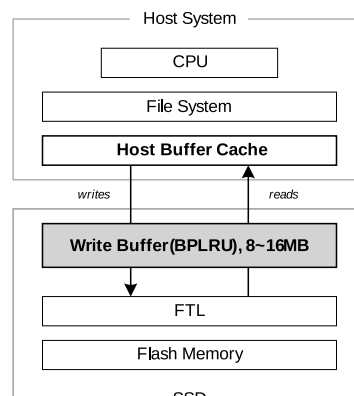


Figure 2: System configuration. BPLRU, the proposed buffer management scheme, is applied to RAM buffer inside SSDs.

cache policy, CFLRU can be applied to reduce the number of write requests by trading off the number of reads. We do not assume any special host side cache policy in this paper.

BPLRU combines three key techniques, which are described in separate subsections: block-level LRU management, page padding, and LRU compensation.

3.1 Block-level LRU

BPLRU manages an LRU list in units of blocks. All RAM buffers are grouped in blocks that have the same size as the erasable block size in the NAND flash memory. When a logical sector cached in the RAM buffer is accessed, all sectors in the same block range are placed at the head of the LRU list. To make free space in the buffer, BPLRU chooses the least recent block instead of a sector, and flushes all sectors in the victim block. This block-level flushing minimizes the log block attaching cost in log-block FTL [17].

Figure 3 shows an example of the BPLRU list. In the figure, eight sectors are in the write buffer, and each block contains four sectors. When sector 15 is written again, the whole block is moved to the head of the LRU list. Thus sector 12 is at the front of the LRU list even though it has not been recently accessed. When free space is required, the least recently used block is selected as a victim block, and all sectors in the victim block are flushed from the cache at once. In the example, block 1 is selected as the victim block, and sectors 5 and 6 are flushed.

Table 1 shows a more detailed example. It assumes that only two log blocks are allowed, and eight sectors can reside in the write buffer. In this example, 14 highly scattered sectors are written in this order: 0, 4, 8, 12, 16, 1, 5, 9, 13, 17, 2, 6, 10, and 14. Because no dupli-

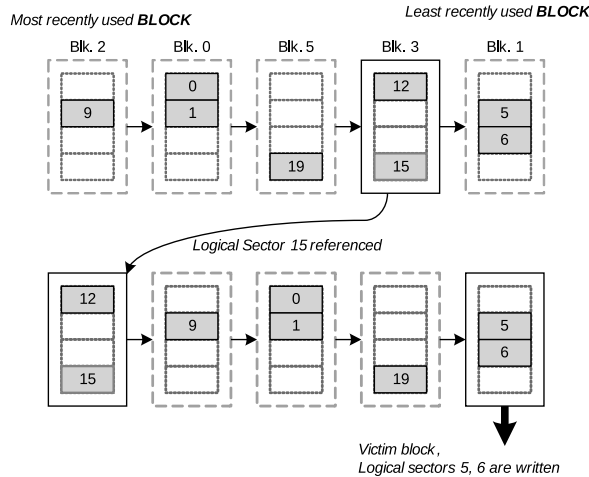


Figure 3: An example of Block-level LRU. When sector 15 is accessed, sector 12 is also moved to the front of the list because sector 12 is in the same block with sector 15.

cated access exists in the sequence, the LRU cache does not influence the write pattern, and these writes evoke 12 merges in the FTL. If we apply block-level LRU for exactly the same write pattern, we can reduce the merge counts to seven.

If write requests are random, the cache hit ratio will be lower. Then, the LRU cache will act just like the FIFO queue and not influence the performance. BPLRU, however, can improve write performance even if the cache is not hit at all. If we calculate the cache hit rate for the block level, it may be greater than zero even though the sector level cache hit rate is zero. BPLRU minimizes merge cost by reordering the write sequences in the erasable block unit of NAND flash memory.

3.2 Page Padding

Our previous research [16] proposed the page padding technique to optimize the log-block FTL algorithm. In the method, we pad the log block with some clean pages from the data block to reduce the number of *full merges*. Here we apply the same concept for our buffer management scheme.

BPLRU uses a page padding technique for a victim block to minimize the buffer flushing cost. In log-block FTL, all sector writes must be performed to log blocks, and the log blocks are merged with data blocks later. When a log block is written sequentially from the first sector to the last sector, it can simply replace the associated data block with a *switch merge* operation. If a victim block flushed by BPLRU is full, then a *switch merge* will occur in the log-block FTL. Otherwise, a relatively expensive *full merge* will occur instead of a *switch merge*.

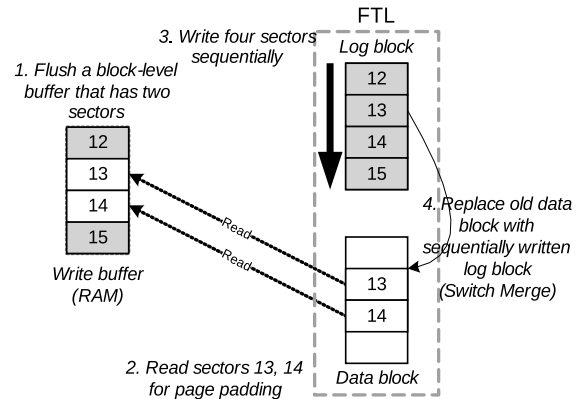


Figure 4: An example of page padding. When Block-level LRU chooses a victim block having sector 12 and sector 15, sector 13 and sector 14 are read from the storage, and four sectors are written back sequentially.

Therefore, BPLRU reads some pages that are not in a victim block, and writes all sectors in the block range sequentially. Page padding may seem to perform unnecessary reads and writes, but it is more effective because it can change an expensive *full merge* to an efficient *switch merge*.

Figure 4 shows an example of page padding. In the example, the victim block has only two sectors (12 and 15), and BPLRU reads sectors 13 and 14 for page padding. Then, four sectors from sectors 12-16 are written sequentially. In log-block FTL, a log block is allocated for the writes, and the log block replaces the existing data block because the log block is written sequentially for all sectors in the block, i.e., a *switch merge* occurs. We show the effectiveness of page padding separately in Section 4.

3.3 LRU Compensation

Because LRU policy is not as effective for sequential writes, some enhanced replacement algorithms such as low inter-reference recency set (LIRS) [9] and adaptive replacement cache (ARC) [19] have been proposed.

To compensate for a sequential write pattern, we used a simple technique in BPLRU. If we know that the most recently accessed block was written sequentially, we estimate that that block has the least possibility of being rewritten in the near future, and we move that block to the tail of the LRU list. This scheme is also important when page padding is used.

Figure 5 shows an example of LRU compensation. BPLRU recognizes that block 2 is written fully sequentially, and moves it to the tail of the LRU list. When more space is needed later, the block will be chosen as a victim block and flushed. We show the effectiveness of LRU

Sector Writes	LRU		Block-level LRU	
	Cache status (8 sectors)	Log block (2 blocks)	Cache status (8 sectors)	Log block (2 blocks)
0, 4, 8, 12, 16	16, 12, 8, 4, 0		[16], [12], [8], [4], [0]	
1, 5, 9	9, 5, 1, 16, 12, 8, 4, 0		[8, 9], [4, 5], [0, 1], [12], [16]	
13	13, 9, 5, 1, 16, 12, 8, 4 → 0	[0]	[12, 13], [8, 9], [4, 5], [0, 1] → 16	[16]
17	17, 13, 9, 5, 1, 16, 12, 8 → 4	[4], [0]	[17], [12, 13], [8, 9], [4, 5], → 0, 1	[0, 1], [16]
2	2, 17, 13, 9, 5, 1, 16, 12 → 8	[8], [4] → M[0]	[2], [17], [12, 13], [8, 9], [4, 5]	
6	6, 2, 17, 13, 9, 5, 1, 16 → 12	[12], [8] → M[4]	[6], [2], [17], [12, 13], [8, 9] → 4, 5	[4, 5], [0, 1] → M[16]
10	10, 6, 2, 17, 13, 9, 5, 1 → 16	[16], [12] → M[8]	[8, 9, 10], [6], [2], [17], [12, 13]	
14	14, 10, 6, 2, 17, 13, 9, 5 → 1	[1], [16] → M[12]	[14], [8, 9, 10], [6], [2], [17] → 12, 13	[12, 13], [4, 5] → M[0, 1]
Buffer flushing by LRU order	14, 10, 6, 2, 17, 13, 9 → 5	[5], [1] → M[16]	[14], [8, 9, 10], [6], [2] → 17	[17], [12, 13] → M[4, 5]
	14, 10, 6, 2, 17, 13 → 9	[9], [5] → M[1]	[14], [8, 9, 10], [6] → 2	[2], [17] → M[12, 13]
	14, 10, 6, 2, 17 → 13	[13], [9] → M[5]	[14], [8, 9, 10] → 6	[6], [2] → M[17]
	14, 10, 6, 2 → 17	[17], [13] → M[9]	[14] → 8, 9, 10	[8, 9, 10], [6] → M[2]
	14, 10, 6 → 2	[2], [17] → M[13]	→ 14	[14], [8, 9, 10] → M[6]
	14, 10 → 6	[6], [2] → M[17]		
	14 → 10	[10], [6] → M[2]		
	→ 14	[14], [10] → M[6]		
	Total merge count	12		7

Note: [], →, M mean a block boundary, flushing, merge operation in FTL, respectively

Table 1: Comparison of LRU and Block-level LRU. All sectors in the example write sequence are written only once. Thus, LRU cache acts just like FIFO queue, and does not influence write performance. Meanwhile, Block-level LRU can reduce the number of merges in FTL by reordering the write sequences in the erasable block unit of NAND flash memory.

compensation in Section 4 with a sequential workload.

3.4 Implementation Details

Since BPLRU is for the RAM buffer inside an SSD, we must design BPLRU to use as little CPU power as possible. The most important part of LRU implementation is to find the associated buffer entry quickly because searching is required for every read and write request.

For this purpose, we used a two-level indexing technique. Figure 6 illustrates the data structure of BPLRU. There are two kinds of nodes, a sector node and a block header node, but we designed our system to share only one data structure to simplify memory allocation and free processes. Free nodes are managed by one free node list, and a free node can be used for a block header node or a sector node.

The members of the node structure are defined as follows: Two link points for LRU or a free node list (nPrev, nNext), block number (nLbn), number of sectors in a block (nNumOfSct), and sector buffer (aBuffer). When a node is used as a sector node, aBuffer[] contains the contents of the writing sector, while it is functioning as a

secondary index table that points to its child sector nodes when the node is used as a block header node. That is, every block header node has its second-level index table.

With this two-level indexing technique, we can find the target sector node using just two index references. The memory for the block header nodes should be regarded as the cost for the fast searching. To find a sector node with a sector number, first we calculate the block number by dividing the sector number by the number of sectors per block, N . Then, the first-level block index table is referenced with the calculated block number. If no associated block header exists, the sector is not in the write buffer. If a block header node exists, then we check the secondary index table in the block header node with the residue of dividing the sector number by N .

4 Evaluation

To evaluate BPLRU, we used both simulation and experiments on a real hardware prototype to compare four cases: no RAM buffer, LRU policy, FAB policy, and the BPLRU method.

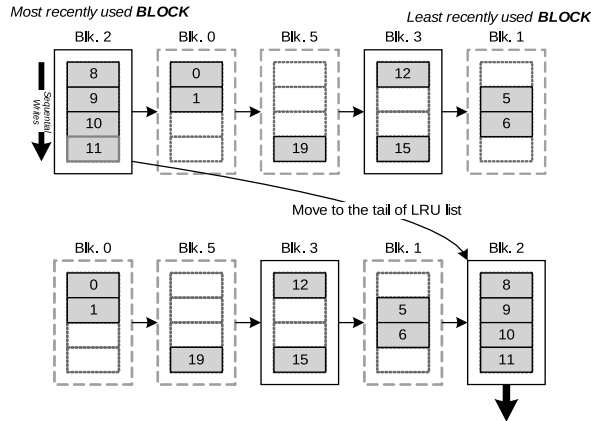


Figure 5: An example of LRU compensation. When sector 11 is written, BPLRU recognizes that block 2 is written fully sequentially, and moves it to the tail of the LRU list.

4.1 Collecting Write Traces

For our experiments, we collected write traces from three different file systems: NTFS, FAT16, and EXT3. Since we were interested in the random write performance, we excluded read requests from the traces. We attached a new hard disk to the test computer, and used a 1-GB partition for collecting write traces; our prototype hardware had 1-GB capacity limitation.

We used *Blktrace* [1] on Linux and *Diskmon* [24] on Windows XP to collect the traces. For Windows Vista, we used *TraceView* [20] in the Windows Driver Development Kit to collect write traces including buffer flush command.

We used nine different work tasks for our experiments.

W1: MS Office 2003 Installation. We captured the write traces while installing MS Office 2003 with the full options on Windows XP using NTFS. However, because we used a separate partition for the installation, some writes for Windows DLL files were missed in our traces.

W2: Temporary Internet Files. We changed the position of the temporary Internet files folder to a 1-GB partition formatted with NTFS. We then collected traces while surfing the Web with Internet Explorer 6.0 until we filled the folder with 100 MB of data spread across several small files.

W3: PCMark05. *PCMark05* [5] is one of the most widely used benchmark programs for Windows. It consists of several tests, and subsets of the test can be performed separately. We performed the HDD test and

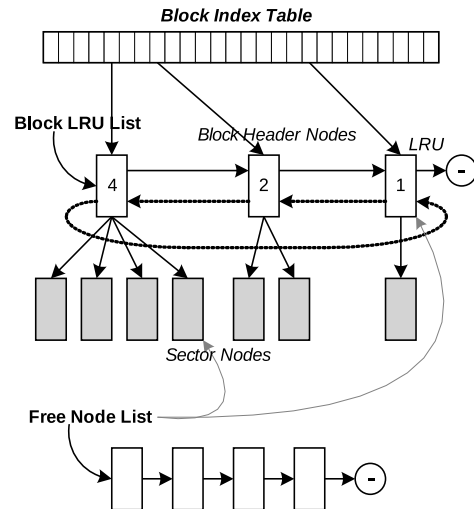


Figure 6: Data structure of BPLRU. For fast buffer lookup, two-level indexing is used. The first-level table is indexed by block numbers. Each entry in the table points to the block header if there are some buffers belonging to the block. Otherwise it has a null value. Each block header has second-level index table for all sectors in the block.

traced the write requests on NTFS in Windows XP.

W4: Iometer. We used *Iometer* [7] on NTFS to produce pure uniformly distributed random write accesses. *Iometer* creates a large file with the size of the complete partition, and then overwrites sectors randomly.

W5: Copying MP3 Files. In this task, we copied 90 MP3 files to the target partition and captured the write traces. This task was chosen to examine the effect of the write buffer when writing large files. Even though the goal of BPLRU is to enhance random write performance, it should not be harmful for sequential write requests because sequential patterns are also very important. The average file size was 4.8 MB. We used the FAT16 file system since it is the most popular in mobile multimedia devices such as MP3 players and personal media players.

W6: P2P File Download. We used *emule* [11] to capture write traces of a peer-to-peer file download program. Since *emule* permits configuring the positions of temporary and download folders, we changed the temporary folder position to the target partition. We then downloaded a 634-MB movie file and captured the write traces. We used the FAT16 file system, and this pattern can be considered as the worst case. Small parts of a file are written almost randomly to the storage because the peer-to-peer program downloads different parts concurrently from numerous peers.

Operation	Time(us)
256-KB Block Erase	1500
2KB-Page Read	50
2KB-Page Write	800
2KB-Data Transfer	50

Table 2: Timing parameters for simulation. These are typical timing values that are shown in the datasheet of MLC NAND flash memory.

W7: Untar Linux Sources. We downloaded *linux-2.6.21.tar.gz* from the Internet and extracted the source files to the target partition, which was formatted with the EXT3 file system. After the extraction, we had 1,285 folders and 21,594 files, and the average file size was 10.7 KB.

W8: Linux Kernel Compile. With the sources of *linux-2.6.21*, we compiled the Linux kernel with the default configuration and collected write traces.

W9: Postmark. We chose *Postmark* because it is widely used for evaluating the performance of I/O subsystems. The benchmark imitates the behavior of an Internet mail server and consists of a series of transactions. In each transaction, one of four operations (file creation, deletion, read, or write) is executed at random. We collected the write traces from all three file systems. Because *Postmark* can be compiled and run under Windows as well as Linux, we used it to compare the performance of BPLRU among the different file systems.

4.2 Simulation

4.2.1 Environment

For our simulation, we assumed the log-block FTL algorithm and a 1-GB MLC NAND flash memory with 128 2-KB pages in each block. The log-block FTL was configured to use seven log blocks. We ignored the map block update cost in the FTL implementation because it was trivial. We simulated the write throughput and the required number of block erase operations while varying the RAM buffer size from 1 to 16 MB.

We used the parameters of Table 2 for the performance calculation. Note that these values are typical of those provided by MLC NAND flash memory data sheets [25]. Some differences may exist in real timing values.

4.2.2 Performance and Erase Count

The simulation results for W1 through W8 are shown in Figure 7 through Figure 14, respectively.

For the *MS Office Installation* task (W1), BPLRU exhibited a 43% faster throughput and 41% lower erase count than FAB for a 16-MB buffer.

For the W2, the BPLRU performance was slightly worse than FAB for buffers less than 8 MB. However, the performance improved for buffer sizes larger than 8 MB, and the number of erase operations for BPLRU was always less than for FAB. The result of *PCMark05* (W3) test (Figure 9) was very similar to the *Temporary Internet Files* (W2) test.

The result of the *Iometer* test (W4) was different from all the others. FAB showed better write performance than BPLRU, and the gap increased for bigger buffer cache sizes. This is because of the difference in the victim selection policy between FAB and BPLRU. BPLRU uses the LRU policy, but no locality exists in the write pattern of *Iometer* because it generates pure random patterns. Therefore, considering the reference sequence is a poorer approach than simply choosing a victim block with more utilized sectors like FAB. However, due to page-padding, BPLRU exhibits lower erase counts.

Figure 11 shows the effect of BPLRU for a sequential write pattern (W5). While the result of the peer-to-peer task (W6)(figure 12) illustrates the generally poor performance of flash storage for random writes, it does show that BPLRU can improve the performance of random write requests significantly. We can see that FAB requires more RAM than BPLRU to get better performance.

The two results with the EXT3 file system (Figures 13 and 14) demonstrate the benefits of BPLRU. It provides about 39% better throughput than FAB in the Linux source untar case, and 23% in the Linux kernel compile task with a 16-MB buffer.

4.2.3 File System Comparison

To show the effects of BPLRU on different file systems, we used the *Postmark* benchmark for all three. We can see in Figure 15 that BPLRU exhibits fairly good throughput for all three file systems.

4.2.4 Buffer Flushing Effect

In our simulation, we assumed that no SCSI or SATA buffer flush command was ever sent to the device. However, in practice, file systems use it to ensure data integrity. Naturally, the inclusion of the flush command will reduce the effect of write buffering and degrade BPLRU performance.

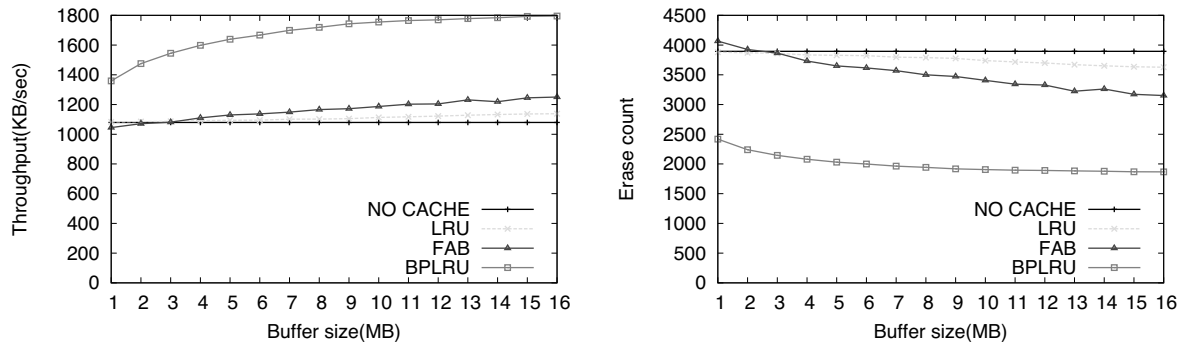


Figure 7: W1 MS Office 2003 installation (NTFS).

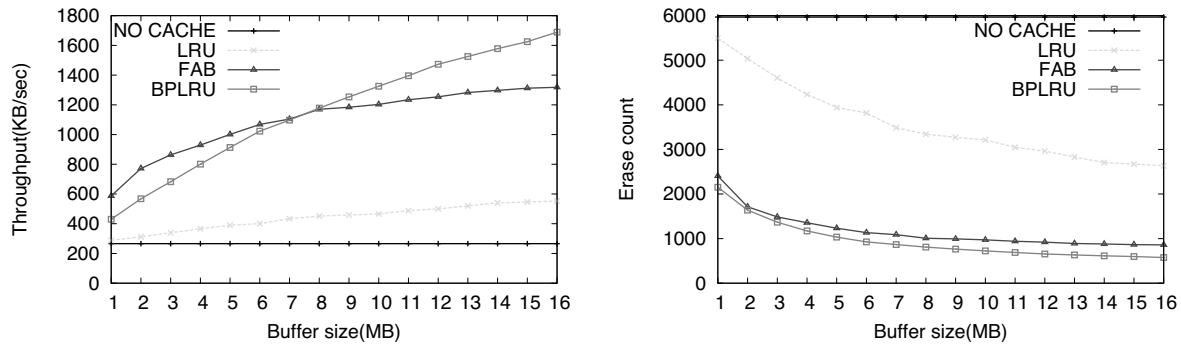


Figure 8: W2 Temporary internet files of Internet Explorer (NTFS).

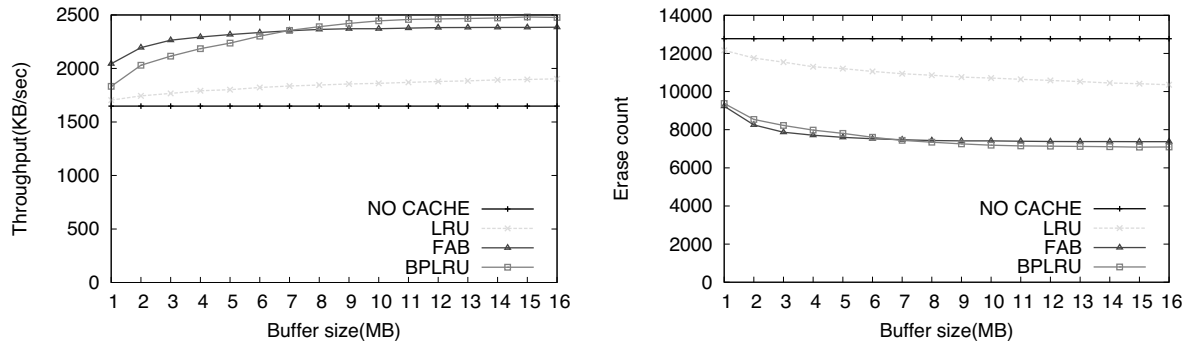


Figure 9: W3 HDD test of PCMark05 (NTFS).

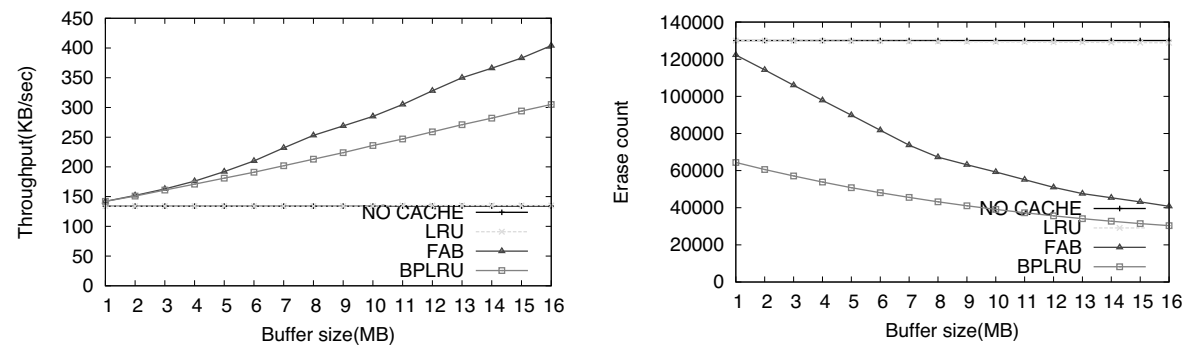


Figure 10: W4 Random writes by Iometer (NTFS).

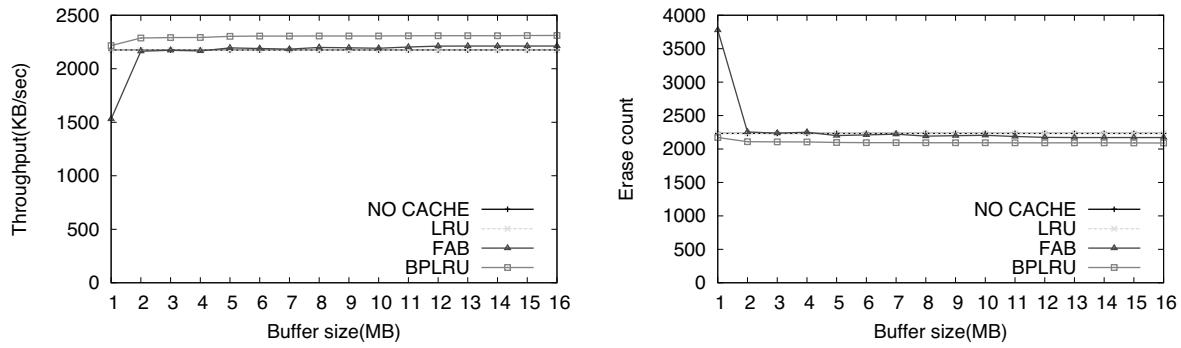


Figure 11: W5 Copy MP3 files (FAT16).

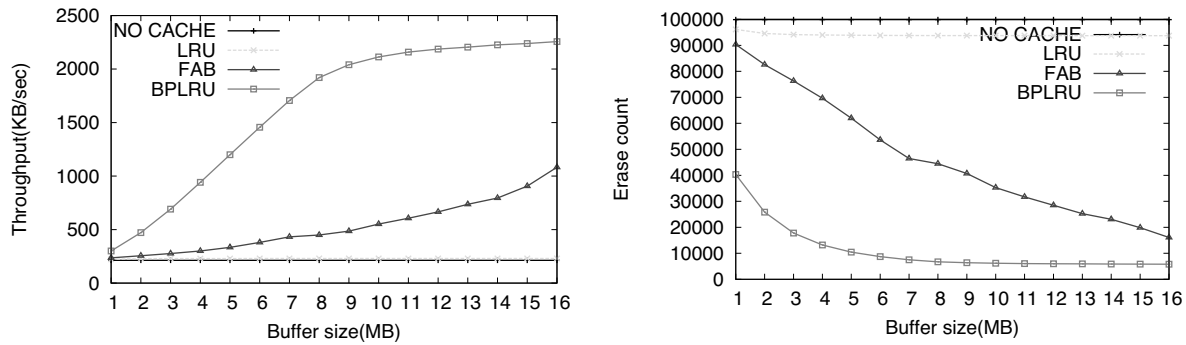


Figure 12: W6 634-MB file download by P2P program, emule (FAT16).

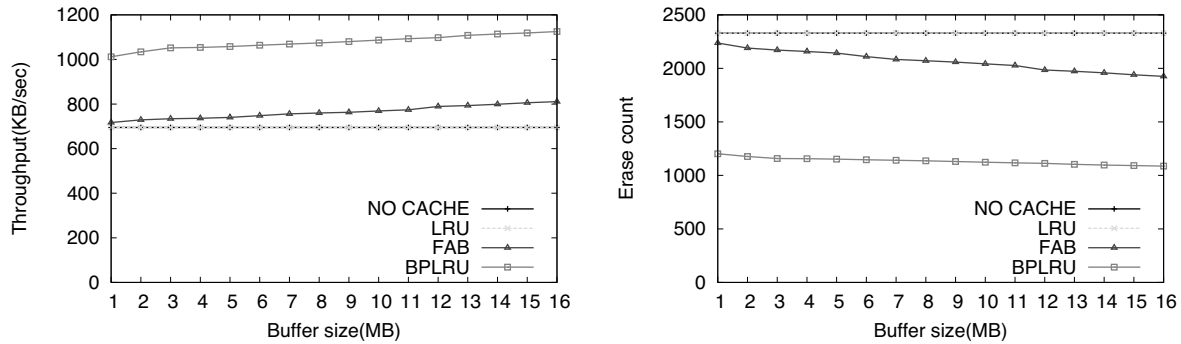


Figure 13: W7 Untar linux source files from linux-2.6.21.tar.gz (EXT3).

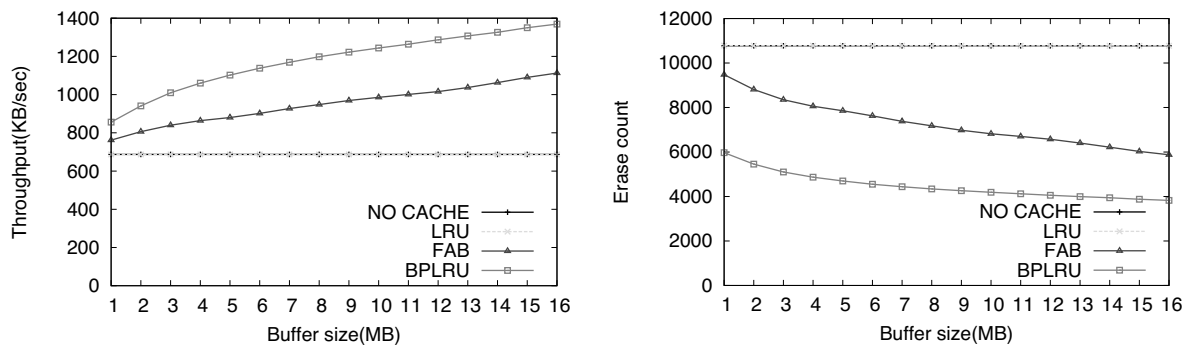


Figure 14: W8 Kernel compile with linux-2.6.21 sources (EXT3)

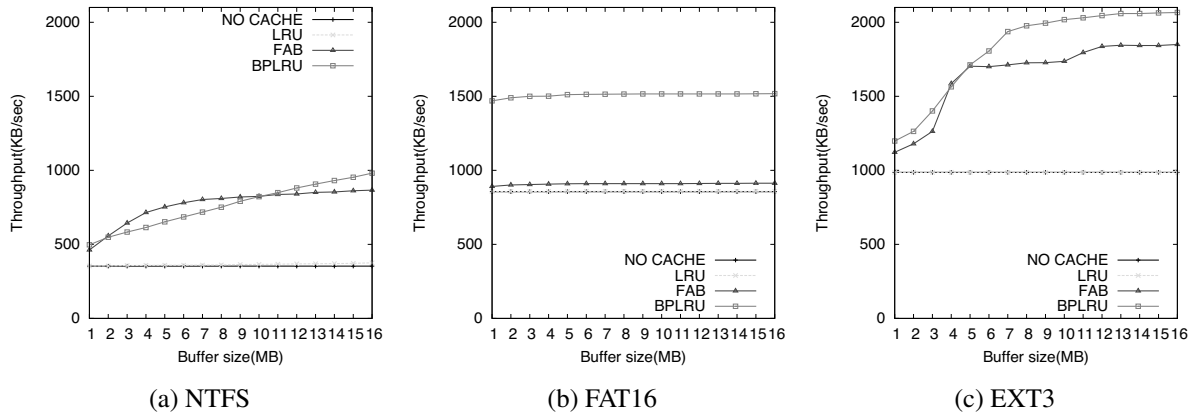


Figure 15: Results of Postmark (W9) on three different file systems. BPLRU shows fairly good performance compared with LRU and FAB. Especially BPLRU shows the biggest performance gain for FAT16.

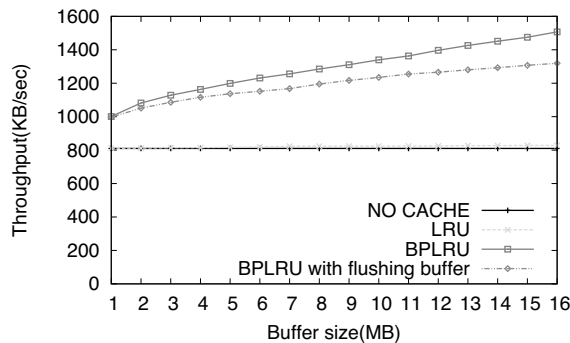


Figure 16: Buffer flushing effect. The throughput of BPLRU is reduced by about 23

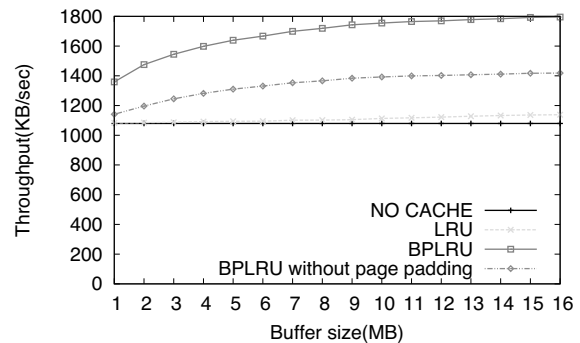


Figure 17: Page padding effect. The throughput for MS Office 2003 installation is reduced by about 21

To collect write traces including buffer flush command, we had to use Windows Vista since the *Trace-View* utility does not trace this command for Windows XP. However, we discovered that for some unknown reason, Windows does not send the buffer flush command to a secondary hard disk.

Because of this, we traced the flush command on the primary C: drive to determine approximately how much it decreases the performance. As the simulation results in Figure 16 show, buffer flushing with a 16-MB buffer reduces the throughput by approximately 23%.

4.2.5 Detailed BPLRU Analysis

We also tested the benefits of page padding and LRU compensation on a subset of the experiments previously described. Figure 17 shows the effect of page padding with the simulation results for the write traces from the MS Office 2003 installation, and Figure 18 shows the effect of LRU compensation for the traces from copying MP3 files on the FAT file system. Page padding and LRU

compensation enhance the throughput by about 26% (16 MB RAM) and 10% (1 MB RAM), respectively.

4.3 Real Hardware Prototype

We also implemented and tested three algorithms on a real prototype based on a target board with an ARM940T processor (Figure 19). It has 32 MB of SDRAM, a USB 2.0 interface, two NAND sockets, and an error correction code hardware engine for MLC and SLC NAND flash memory. We used a 200-MHz clock and a 1-GB MLC NAND flash memory whose page size was 2 KB. Since 128 pages were in each block, the block size was 256 KB.

We used our prototype to implement a USB mass storage (UMS) disk. We used the log-block FTL algorithm because of its wide commercial use. We configured it for seven log blocks and used an 8-MB RAM cache for three algorithms: LRU, FAB, and BPLRU. We used a 2.4-GHz Pentium 4 PC with Windows XP Professional, 512 MB

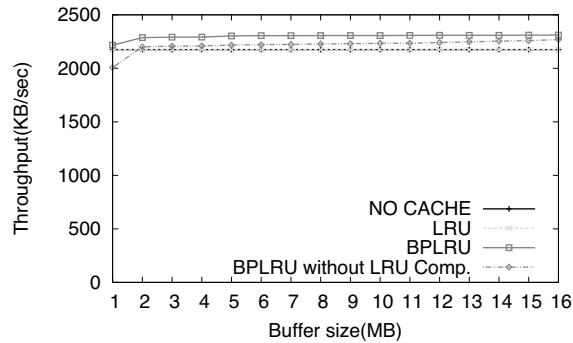


Figure 18: LRU compensation effect. The throughput of BPLRU for copying MP3 files on FAT16 is reduced by approximately 9

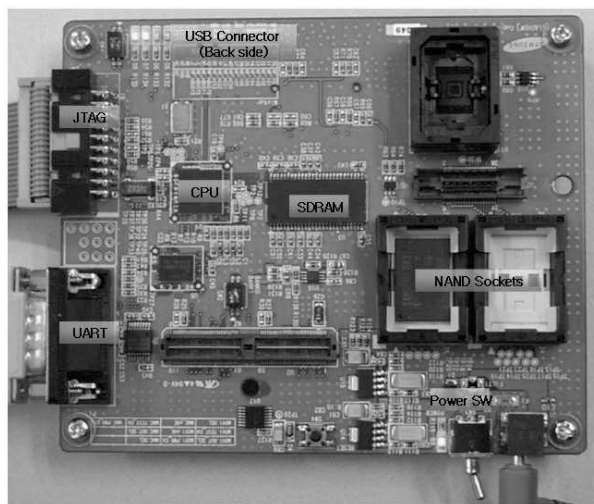


Figure 19: Prototype flash storage board. We implemented USB mass storage device with 1-GB MLC NAND flash memory and 8-MB RAM buffer.

of RAM, and a USB 2.0 interface as a host machine.

We designed a simple application to replay write traces to the UMS disk. After attaching our prototype UMS disk to the host computer, the application replayed the write traces and measured the time automatically. When all traces were written to the UMS disk, our application sent a special signal to the UMS disk to flush its RAM buffer to FTL and report the total elapsed time in seconds including the RAM flushing time. During the test, the trace replay application wrote sectors by directly accessing the Windows device driver, bypassing the Windows file system and buffer cache. Therefore the experimental results were not affected by the file system and the buffer cache of the host.

Figure 20 compares the simulation and the prototype experimental results for six tasks. Some slight differences are evident. For the MP3 copying task, the pro-

totype results show that using the RAM buffer is worse than not using it. Also, the performance of BPLRU in the simulation is slightly better than in our prototype system. The reason for this is that we did not include the RAM copy cost in our simulation, but simply derived the performance from the operation counts of page reads, page writes, and block erasures. In addition, the operation time of NAND flash memory was not constant in our real-target experiments. For example, we used 1.5 ms for the block erase time because this value is given by data sheets as a typical time. However, the actual time may depend on the status of the block on the actual board. Even so, the overall trends were very similar between the prototype experiment and the simulation results.

5 Conclusions

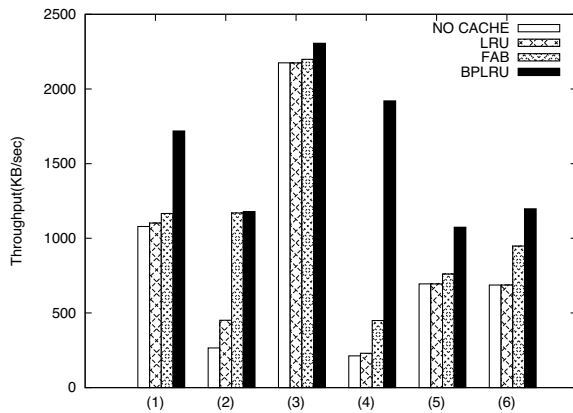
We demonstrated that a certain amount of write buffer in a flash storage system can greatly enhance random write performance. Even though our proposed BPLRU buffer management scheme is more effective than two previous methods, LRU and FAB, two important issues still remain. First, when a RAM buffer is used in flash storage, the integrity of the file system may be damaged by sudden power failures. Second, frequent buffer flush commands from the host computer will decrease the benefit of the write buffer using BPLRU.

Dealing with power failures is an important operational consideration for desktop and server systems. We are considering different hardware approaches to address this issue as an extension of our current work. A small battery or capacitor may delay shutdown until the RAM content is safely saved to an area of flash memory reserved for the purpose; an 8-MB buffer could be copied to that space in less than 1 s. Also, we may use non-volatile magnetoresistive RAM or ferroelectric RAM instead of volatile RAM.

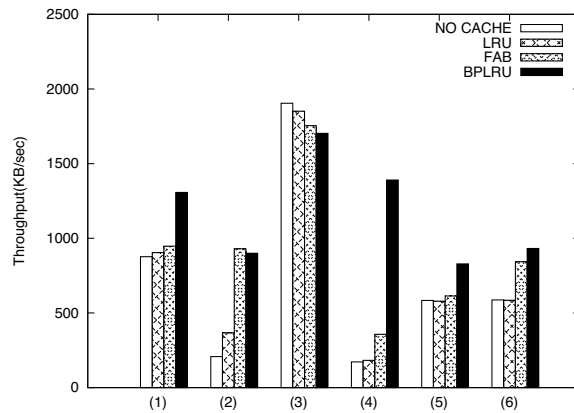
Our study focused on a write buffer in the storage device. However, the BPLRU concept could be extended to the host side buffer cache policy. We may need to consider read requests with a much bigger RAM capacity. At that point, an asymmetrically weighted buffer management policy will be required for read and write buffers, such as CFLRU or LIRS-WSR [12]. This is also a subject for future research.

6 Acknowledgments

We thank Dongjun Shin, Jeongeun Kim, Sun-Mi Yoo, Junghwan Kim, and Kyoungil Bahng for their frequent assistance and valuable advice. We also like to express our deep appreciation to our shepherd, Jiri Schindler, for his helpful comments on how to improve the paper.



(a) Simulation Results



(b) Prototype Experimental Results

Figure 20: Results comparison between simulation and prototype results for six workloads. (1) MS Office Installation (NTFS), (2) Temporary Internet Files (NTFS), (3) Copying MP3 Files (FAT16), (4) P2P File Download (FAT16), (5) Untar Linux Sources (EXT3), and (6) Linux Kernel Compile (EXT3)

References

- [1] Jens Axboe. Block IO Tracing. <http://www.kernel.org/git/?p=linux/kernel/git/axboe/blktrace.git;a=blob;f=README>.
- [2] Amir Ban. Flash File System, 1993. United States Patent, No 5,404,485.
- [3] Fred Douglass, Ramon Caceres, M. Frans Kaashoek, Kai Li, Brian Marsh, and Joshua A. Tauber. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation*, pages 25–37, 1994.
- [4] Douglas Dumitru. Understanding Flash SSD Performance. Draft, <http://www.storagesearch.com/easyco-flashperformance-art.pdf>, 2007.
- [5] Futuremark Corporation. PCMARK'05. <http://www.futuremark.com/products/pcmark05/>.
- [6] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. White Paper, <http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf>, 1998.
- [7] Iometer Project, iometer-[user—devel]@lists.sourceforge.net. Iometer Users Guide. <http://www.iometer.org>.
- [8] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST'05: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.
- [9] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 31–42, New York, NY, USA, 2002. ACM.
- [10] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: flash-aware buffer management policy for portable media players. *Consumer Electronics, IEEE Transactions on*, 52(2):485–493, 2006.
- [11] John, Hendrik Breitkreuz, Monk, and Bjoern. eMule. <http://sourceforge.net/projects/emule>.
- [12] Hoyoung Jung, Kyunghoon Yoon, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. LIRS-WSR: Integration of LIRS and writes sequence reordering for flash memory. *Lecture Notes in Computer Science*, 4705:224–237, 2007.
- [13] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superbblock-based flash translation layer for NAND flash memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pages 161–170, New York, NY, USA, 2006. ACM.
- [14] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.
- [15] Bumsoo Kim and Guiyoung Lee. Method of driving remapping in flash memory and flash memory architecture suitable therefore, 2002. United States Patent, No 6,381,176.
- [16] Hyojun Kim, Jin-Hyuk Kim, ShinHo Choi, HyunRyong Jung, and JaeGyu Jung. A page padding method for fragmented flash storage. *Lecture Notes in Computer Science*, 4705:164–177, 2007.
- [17] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.
- [18] M-Systems. Two Technologies Compared: NOR vs. NAND. White Paper, http://www.dataio.com/pdf/NAND/MSystems/MSystems_NOR_vs_NAND.pdf, 2003.

- [19] Nimrod Megiddo and Dharmendra S. Modha. ARC: a self-tuning, low overhead replacement cache. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.
- [20] Microsoft. Windows Driver Kit: Driver Development Tools, TraceView. <http://msdn2.microsoft.com/en-us/library/ms797981.aspx>.
- [21] Chanik Park, Jeong-Uk Kang, Seon-Yeong Park, and Jin-Soo Kim. Energy-aware demand paging on NAND flash-based embedded storages. In *ISLPED '04: Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 338–343, New York, NY, USA, 2004. ACM.
- [22] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 234–241, New York, NY, USA, 2006. ACM.
- [23] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
- [24] Mark Russinovich. DiskMon for Windows v2.01. <http://www.microsoft.com/technet/sysinternals/utilities/diskmon.mspx>, 2006.
- [25] Samsung Electronics. K9XXG08UXM 1G x8 Bit / 2G x 8 Bit NAND Flash Memory. http://www.samsung.com/global/business/semiconductor/products/flash/Products_NANDFlash.html, 2005.
- [26] SSFDC Forum. SmartMedia Specification. <http://www.ssfdc.or.jp>.

Write Off-Loading: Practical Power Management for Enterprise Storage

Dushyanth Narayanan

Austin Donnelly
Microsoft Research Ltd.

Antony Rowstron

{*dnarayan,austind,antr*}@microsoft.com

Abstract

In enterprise data centers power usage is a problem impacting server density and the total cost of ownership. Storage uses a significant fraction of the power budget and there are no widely deployed power-saving solutions for enterprise storage systems. The traditional view is that enterprise workloads make spinning disks down ineffective because idle periods are too short. We analyzed block-level traces from 36 volumes in an enterprise data center for one week and concluded that significant idle periods exist, and that they can be further increased by modifying the read/write patterns using *write off-loading*. Write off-loading allows write requests on spun-down disks to be temporarily redirected to persistent storage elsewhere in the data center.

The key challenge is doing this transparently and efficiently at the block level, without sacrificing consistency or failure resilience. We describe our write off-loading design and implementation that achieves these goals. We evaluate it by replaying portions of our traces on a rack-based testbed. Results show that just spinning disks down when idle saves 28–36% of energy, and write off-loading further increases the savings to 45–60%.

1 Introduction

Power consumption is a major problem for enterprise data centers, impacting the density of servers and the total cost of ownership. This is causing changes in data center configuration and management. Some components already support power management features: for example, server CPUs can use low-power states and dynamic clock and voltage scaling to reduce power consumption significantly during idle periods. Enterprise storage subsystems do not have such advanced power management and consume a significant amount of power in the data center [32]. An enterprise grade disk such as the Seagate Cheetah 15K.4 consumes 12 W even when

idle [26], whereas a dual-core Intel Xeon processor consumes 24 W when idle [14]. Thus, an idle machine with one dual-core processor and two disks already spends as much power on disks as processors. For comparison, the 13 core servers in our building's data center have a total of 179 disks, more than 13 disks per machine on average.

Saving power in storage systems is difficult. Simply buying fewer disks is usually not an option, since this would reduce peak performance and/or capacity. The alternative is to spin down disks when they are not in use. The traditional view is that idle periods in server workloads are too short for this to be effective [5, 13, 32]. In this paper we present an analysis of block-level traces of storage volumes in an enterprise data center, which only partially supports this view. The traces are gathered from servers providing typical enterprise services, such as file servers, web servers, web caches, etc.

Previous work has suggested that main-memory caches are effective at absorbing reads but not writes [4]. Thus we would expect at the storage level to see periods where all the traffic is write traffic. Our analysis shows that this is indeed true, and that the request stream is write-dominated for a substantial fraction of time.

This analysis motivated a technique that we call *write off-loading*, which allows blocks written to one volume to be redirected to other storage elsewhere in the data center. During periods which are write-dominated, the disks are spun down and the writes are redirected, causing some of the volume's blocks to be off-loaded. Blocks are off-loaded temporarily, for a few minutes up to a few hours, and are reclaimed lazily in the background after the home volume's disks are spun up.

Write off-loading modifies the per-volume access patterns, creating idle periods during which all the volume's disks can be spun down. For our traces this causes volumes to be idle for 79% of the time on average. The cost of doing this is that when a read occurs for a non-off-loaded block, it incurs a significant latency while the disks spin up. However, our results show that this is rare.

Write off-loading is implemented at the block level and is transparent to file systems and applications running on the servers. Blocks can be off-loaded from any volume to any available persistent storage in the data center, either on the same machine or on a different one. The storage could be based on disks, NVRAM, or solid-state memory such as flash. Our current hardware does not have flash or other solid-state devices and hence we used a small partition at the end of each existing volume to host blocks off-loaded from other volumes.

Write off-loading is also applicable to a variety of storage architectures. Our trace analysis and evaluation are based on a Direct Attached Storage (DAS) model, where each server is attached directly to a set of disks, typically configured as one or more RAID arrays. DAS is typical for small data centers such as those serving a single office building. However, write off-loading can also be applied to network attached storage (NAS) and storage area networks (SANs).

A major challenge when off-loading writes is to ensure consistency. Each write request to any volume can be off-loaded to one of several other locations depending on a number of criteria, including the power state and the current load on the destination. This per-operation load balancing improves performance, but it means that successive writes of the same logical block could be off-loaded to different destinations. It is imperative that the consistency of the original volume is maintained even in the presence of failures. We achieve this by persisting sufficient metadata with each off-loaded write to reconstruct the latest version of each block after a failure.

This paper makes two main contributions. First, we show that contrary to conventional wisdom and current practice, idle periods in enterprise workloads can be exploited by spinning disks down, for power savings of 28–36%. Second, we present *write off-loading* as a generic and practical approach that allows further reduction of power consumption in storage systems and also eliminates the spin-up penalty for write requests. In our trace-based evaluation on a rack-mounted testbed, write off-loading enabled energy savings of 45–60%. The performance of all write requests, and 99% of read requests, was equivalent to that when not spinning disks down.

The rest of the paper is organized as follows. Section 2 presents an analysis of block-level traces from an enterprise data center, which motivates write off-loading. Section 3 describes the design and implementation of the write off-loading infrastructure. Section 4 presents an evaluation of write off-loading on a rack-based hardware testbed. Section 5 discusses related work, and Sections 6 and 7 conclude the paper.

Server	Function	#volumes
usr	User home directories	3
proj	Project directories	5
prn	Print server	2
hm	Hardware monitoring	2
rsrch	Research projects	3
prxy	Firewall/web proxy	2
src1	Source control	3
src2	Source control	3
stg	Web staging	2
ts	Terminal server	1
web	Web/SQL server	4
mds	Media server	2
wdev	Test web server	4

Table 1: Data center servers traced (13 servers, 36 volumes, 179 disks)

2 Volume Access Patterns

The traditional view is that spinning disks down does not work well for server workloads [5, 13, 32]. Gurusurthi et al. [13] show that for disk traffic patterns generated by the TPC-C and TPC-H benchmarks, spinning down disks is ineffectual: the periods of idleness are too short. Zhu et al. [32] also use the *cello* block-level volume traces [22] collected from a single file/compute server at HP Labs. These are not necessarily representative of all server workloads in enterprise data centers. Many enterprise servers are less I/O intensive than TPC benchmarks, which are specifically designed to stress the system under test. Enterprise workloads also show significant variation in usage over time, for example due to diurnal patterns.

In order to understand better the I/O patterns generated by standard data center servers, we instrumented the core servers in our building’s data center to generate per volume block-level traces for one week. Table 1 describes the servers that we traced: most of these are typical of any enterprise data center. In total, we traced 36 volumes containing 179 disks on 13 servers.

The data center is air-conditioned and the servers are high-end rack-mounted machines. The default configuration is for each server to have two internal physical disks configured as a RAID-1 array, which is used as the boot volume. Each server is additionally configured with one or more RAID-5 arrays as data volumes: the storage for these is provided using rack-mounted co-located DAS. All the servers run the Windows Server 2003 SP2 operating system. Data on the volumes is stored through the NTFS file system and accessed by clients through a variety of interfaces including CIFS and HTTP.

We believe that the servers, data volumes, and their access patterns are representative of a large number of

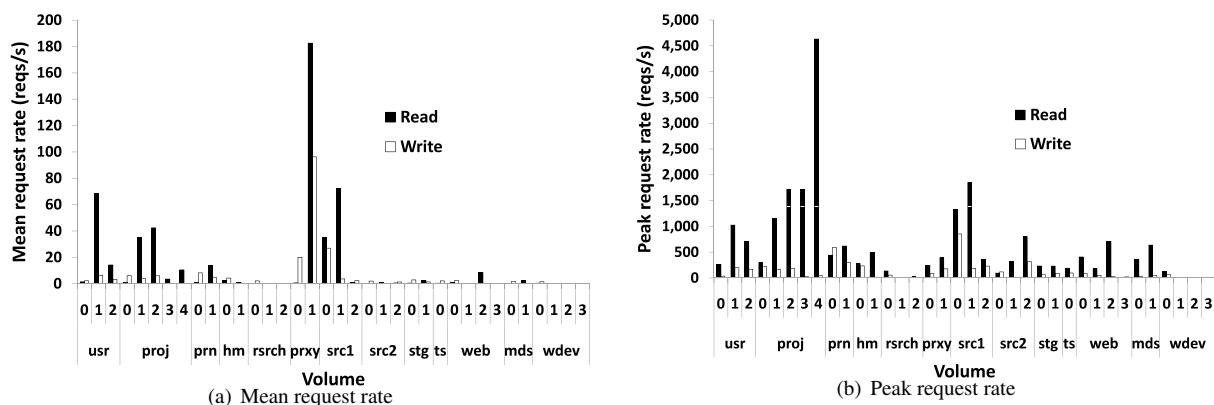


Figure 1: Mean and peak request rates per volume over 7 days

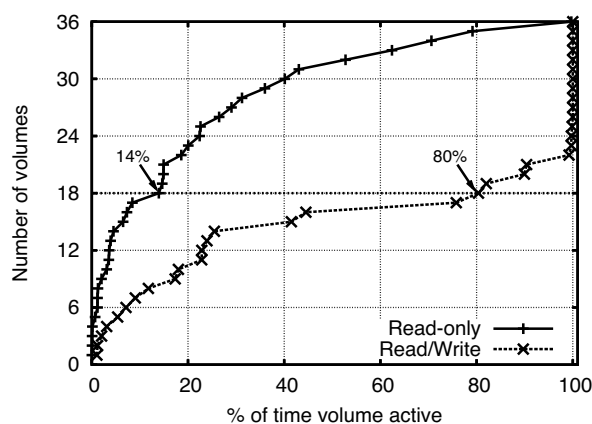


Figure 2: CDF of active time per volume

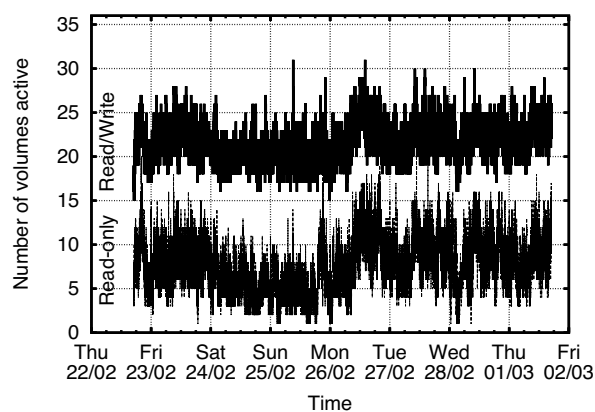


Figure 3: Number of active volumes over time

small to medium size enterprise data centers. Although access patterns for system volumes may be dependent on, for example, the server's operating system, we believe that for data volumes these differences will be small.

The traces are gathered per-volume below the file system cache and capture all block-level reads and writes performed on the 36 volumes traced. The traced period was 168 hours (1 week) starting from 5PM GMT on the 22nd February 2007. The traces are collected using Event Tracing For Windows (ETW) [17], and each event describes an I/O request seen by a Windows disk device (i.e., volume), including a timestamp, the disk number, the start logical block number, the number of blocks transferred, and the type (read or write). ETW has very low overhead, and the traces were written to a separate server not included in the trace: hence we do not believe that the tracing activity perturbed the traced access patterns. The total number of requests traced was 434 million, of which 70% were reads; the total size of the traces was 29 GB. A total of 8.5 TB was read and 2.3 TB written by the traced volumes during the trace period.

Figure 1(a) shows the average read and write request

rate over the entire week for each of the volumes. There is significant variation across volumes, with volumes for the file servers, the source version control servers, and the web proxy all having significant read load. However, many of the volumes such as the research projects server and the test web server have low read and write load. Figure 1(b) shows the peak read and write rates, measured at a 60-second granularity for the 36 volumes. Peak loads are generally much higher than the mean load, indicating that while volumes may be provisioned for a high peak load, most of the bandwidth is unused most of the time.

Overall, the workload is read-dominated: the ratio of read to write requests is 2.37. However, 19 of the 36 volumes have read/write ratios below 1.0; for these volumes the overall read-write ratio is only 0.18. Further analysis shows that for most of the volumes, the read workload is bursty. Hence, intuitively, removing the writes from the workload could potentially yield significant idle periods.

Figure 2 confirms this intuition. It shows a cumulative distribution function across volumes of the number of volumes versus the percentage of time that the volume is active over the course of a week. We show the distribu-

	Mean	Median	99 th pctl	Max
Read/write	21.7 (60%)	22 (61%)	27 (75%)	31 (86%)
Read-only	7.6 (21%)	7 (19%)	15 (42%)	22 (61%)

Table 2: Number of concurrently active volumes: numbers in parentheses show the number of active volumes as a percentage of the total number of volumes (36)

tion both for the original trace (read/write) as well as the trace with the writes removed. In both cases we consider the volume to be idle (i.e., not active) when 60 seconds have elapsed since the last request.

Figure 2 shows that even without removing the writes, there is significant idle time for the volumes. As expected, the write workload has a large impact on the length of the idle periods. When the write load is removed, the mean amount of time a volume is active is only 21%. By contrast, the volume active time in the read/write case is 60% on average. Similarly, the median amount of time a volume is active drops from 80% to 14% when the write load is removed.

Finally, we measure the potential benefit in reducing the peak power consumption of the data center storage by examining the temporal relationship between volumes. Figure 3 shows the number of volumes active over time through the week. We see that removing the writes from the trace significantly reduces the number of concurrently active volumes.

Table 2 shows the mean, median, 99th percentile, and maximum number of volumes concurrently active during the week. These results indicate that simply spinning down when idle can reduce the peak power of the storage subsystem, and that creating longer idle periods by off-loading writes can reduce it even further. Note that the set of active volumes changes over time, and a rarely-active volume might still store a large amount of data or experience a high peak load. Thus we cannot simply save energy by using fewer disks per volume, since we must still provision the volumes for capacity and peak load.

This analysis indicates that there are significant potential power savings in spinning down enterprise data center disks when idle. Further, it shows that efficiently redirecting writes creates even longer periods of idleness leading to substantially higher power savings. This motivated the design of our write off-loading mechanisms.

3 Write Off-Loading

The goal of write off-loading is to utilize periods of write-dominated load to spin disks down and off-load write requests, reverting to normal operation during peri-

ods of read-dominated load. When writes are being off-loaded the aim is to achieve comparable write response times and throughput to using the local volume.

Each volume supporting off-loading has a dedicated *manager*. The manager is entirely responsible for the volume, which we refer to as its *home* volume: it decides when to spin the physical disks up or down, and also when and where to off-load writes. Off-loaded blocks are only temporarily off-loaded and the manager is also responsible for reclaiming blocks previously off-loaded. To achieve all these tasks, the manager needs to intercept all read and write requests to its home volume.

When a manager decides to off-load a block, it selects one or more *loggers* to store it temporarily. Each logger instance requires a small area of persistent storage, which is used exclusively to store off-loaded blocks and metadata until they are reclaimed by a manager or no longer required. The persistent storage could be a disk, NVRAM or solid-state memory such as flash, depending on what is available on each server; the logger's data layout should be optimized for the particular type of storage. Our current implementation uses only disk-based loggers.

The set of loggers that a manager uses is configurable. It is important that the loggers used by a manager offer the same or better failure properties as the home volume. It is also possible to configure the manager so that it will only off-load blocks to loggers residing on the same server as itself, in the same rack, or across the entire data center. We have evaluated write off-loading at both a server and rack granularity. Current off-the-shelf gigabit networking makes the rack granularity feasible, with low network overhead and good performance. Server-granularity off-loading is feasible at any network speed since off-load traffic does not go over the network.

In the rest of this paper, we refer to a volume as being *active* if its disks are spinning and I/O operations are being performed on it. If the disks are spinning but no I/O operations are being performed, we refer to the volume as being *idle*: in this state the disk spindles continue to use energy even though they are doing no work. Finally, if the volume's disks are spun down, we refer to the volume as being in the *standby* state. We assume that all the disks belonging to a volume are always in the same state, since all the power management strategies considered in this paper operate on entire volumes at a time.

When we refer to a manager or logger component being in standby, we mean that the volume used by that component has transitioned to the standby state. When a manager goes into standby, it will force loggers sharing the same physical disks to go into the standby state. The manager will then off-load writes to loggers that are not in the standby state. Note that loggers using solid-state memory or NVRAM would never enter the standby state.

3.1 Detailed Design

Loggers. Conceptually the logger's role is simple: it temporarily stores blocks. Loggers support the following remote operations: *write*, *read*, *invalidate*, and *reclaim*. A write consists of persisting the provided blocks and metadata. The metadata consists of the source manager identity, a range of logical block numbers (LBNs), and a version number. A read returns the latest stored versions of the requested blocks. An invalidate request specifies a set of blocks and versions that are no longer required. To ensure consistency, the invalidate request explicitly includes version information, and the logger marks the corresponding versions as invalid. The logger can then lazily garbage collect the space used to store the invalidated data and metadata. A reclaim request is like a read, except that no block range is specified: the logger can return any valid block range it is holding for the requesting manager. Invalidates and reclaims are non-latency-critical operations; reads and writes are latency-critical but reads are expected to be rare. Hence loggers are optimized for the performance of writes.

Our current implementation uses a log-based on-disk layout. This means that writes have good locality; both data and metadata are written with a single I/O to the current head of the log. Log compaction and other maintenance tasks are done in the background with low priority. Metadata about the valid blocks stored for each manager, their versions, and their location in the log are cached in main memory for fast access.

Each logger uses a small partition at the end of an existing volume to persist data and metadata. This avoids the need to dedicate additional storage for off-loading. The remainder of the volume functions as before, and could have an associated manager to enable off-loading. In general a volume might host zero or more managers and zero or more loggers, on distinct partitions but on the same set of physical disks. In our evaluation we run with a typical configuration for a data volume: one manager and one logger with the latter using a small partition at the end.

Managers. The manager controls the off-loading of blocks, deciding when to off-load blocks and when to reclaim them. It is also responsible for ensuring consistency and performing failure recovery. To achieve this, each manager maintains persistently the identities of a set of loggers with which it interacts, referred to as the *logger view*. It also maintains two in-memory data structures, as shown in Figure 4. The *redirect cache* stores, for each block off-loaded, the block's LBN, the identity of the logger storing the current data for the block and the corresponding version number. Version numbers are unique monotonically increasing 64-bit quantities, which

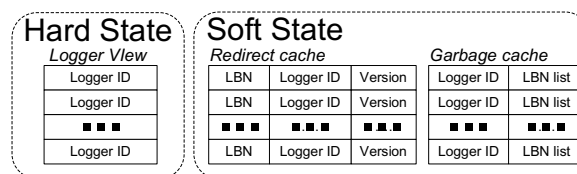


Figure 4: Manager data structures.

ensure that the manager can identify the last written version of any block during failure recovery. The *garbage cache* stores the location of old versions of blocks. In the background, the manager sends invalidation requests for these versions; when these are committed by the logger they are removed from the garbage cache.

The manager intercepts all read and write requests sent to the home volume. For a read request, it first checks the redirect cache for existing logged versions. If none is found, the read is serviced locally from the home volume, causing it to transition from standby to active if necessary. Otherwise the request is dispatched to the logger identified as having the latest version of the block. Multi-block reads are split as required, to fetch data from the home volume and/or one or more loggers.

For a write request, the manager off-loads the write to a logger if the home volume is in standby. It also off-loads the write if there are currently logged versions of any of the blocks, to ensure that the new version is persistently recorded as the latest version. Writes that are not off-loaded are sent directly to the home volume.

To off-load a write, the manager probes the loggers in its logger view: this is currently done using subnet broadcast for efficiency. Each logger replies with a set of metrics including the power state of the logger's volume, its queue length, the amount of available space, etc. The manager ranks the loggers using these metrics and selects one to off-load the write to. When the write is committed and acknowledged by the logger, the manager updates its redirect cache with the latest version and moves any older versions to the garbage cache.

When the home volume is idle, the manager reclaims off-loaded blocks from loggers in the background and writes them to the home volume. After the reclaimed blocks are written to disk, the manager sends invalidation requests to the appropriate loggers. To ensure correct failure recovery, the latest version of a block is invalidated only after all older versions have been invalidated. The background reclaim and invalidation ensure that all blocks will eventually be restored to the home volume and that logger space will eventually be freed.

Finally, the manager controls state transitions to and from standby for the home volume. The manager monitors the elapsed time since the last read and the last write; if both of these have passed a certain threshold,

it spins the volume down and off-loads all subsequent writes. The volume spins up again when there is a read on a non-off-loaded block, or when the number of off-loaded blocks reaches a limit (to avoid off-loading very large amounts of data). Before putting the volume into standby, the manager first ensures that there is at least one logger in its logger view that is using a set of disks different from its own and that is not currently in standby. This ensures that any future writes to the home volume can be off-loaded by the manager without waiting for disks to spin up. If there are no such loggers, then the manager does not spin down, but periodically probes its logger set for any change in their status.

This design is optimized for the common case: during periods of intense activity, the home volumes will be in the active state, and all I/Os will be local, except for a small number of requests on blocks that are currently off-loaded. During periods of low, write-dominated load, we expect that the home volume will be in standby and writes will be successfully off-loaded to a logger.

Uncommon cases are handled through fall-back strategies. For example, if the manager cannot find any available loggers, it spins up the home volume in the background, and retries the request until a logger is found or the home volume is spun up. If a volume needs to be taken off-line (say for maintenance) then the manager spins it up, as well as all volumes that it depends on or that depend on it. It then forces blocks to be reclaimed until the volume has all its own blocks and none of any other's, i.e., its state is restored as if no off-loading had occurred.

Write off-loading can mask the performance impact of spinning up disks for write requests. For read requests on spun-down disks we cannot mask the spin-up delay. For some applications this large delay (10–15 seconds) will be unacceptable even if rare: write off-loading should not be enabled on the volumes that these applications use.

3.2 Failure Resilience

Enterprise storage is expected to provide consistency and durability despite transient failures such as reboots as well as single-disk permanent failures. At the volume level, the failure resilience with off-loading is the same as that without. However, off-loading can create failure dependencies between managers and loggers. With off-loading at the rack or data center level, a manager on machine A could off-load blocks to a logger on machine B: if machine B suffers a failure, then the off-loaded blocks would become unavailable on machine A until machine B was brought on-line again.

This problem can be solved by off-loading each block to multiple independent loggers. With k -way logging, a manager can tolerate up to $k - 1$ failures in its logger

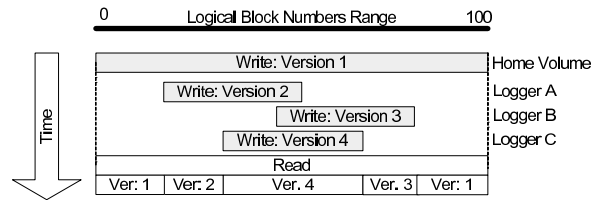


Figure 5: Consistency across loggers example

view. Given the high availability and reliability of enterprise servers, we do not think k -way logging would be required in most cases.

Write off-loading guarantees both consistency and durability across failures. We achieve durability by acknowledging writes only when both data and metadata have been reliably persisted, i.e., we do not employ write-back caching of any form. Consistency is achieved by using versioned metadata to mark the latest version of a block. When a read is performed for a range of blocks, it is quite possible that the required blocks are distributed over multiple loggers as well as the home volume, as shown in Figure 5. The manager uses the version information to ensure that the applications using the volume see a consistent view of the stored data. We also add a checksum to the metadata to ensure that partial writes are correctly detected on failure recovery.

If one or more machines reboot due to, say, a power failure, all the loggers recover concurrently by scanning their persistent logs to reconstruct their soft state. Each manager can be brought on-line when all the loggers in its logger view are on-line. A manager recovers its soft state (the redirect cache and garbage cache) by requesting information about all blocks stored for it from each logger in its logger view. To optimize the common case of a clean shutdown/reboot of a server, the manager writes the soft state to a small metadata partition during shutdown; this allows managers to restart after a clean shutdown without any network communication.

It is important that a manager's logger view be restricted to loggers which have the same or higher failure resilience as the home volume. Otherwise, when blocks are off-loaded, they will not have the same failure resilience as non-off-loaded blocks. If the storage uses standard solutions such as RAID-1 or RAID-5 for all volumes, then this property will be ensured, and off-loading will provide the same resilience to single disk failures as standard RAID solutions.

When a logger experiences a single-disk failure, it pushes all off-loaded blocks to other loggers or the appropriate manager, which should typically take seconds to minutes. This reduces the risk of losing off-loaded blocks due to multiple disk failures; the risk can be further reduced if desired by using k -way logging.

4 Evaluation

Section 2 presented a trace-driven analysis showing the potential benefits of write off-loading. This analysis was based on block-level traces of enterprise data center workloads. In this section we evaluate write off-loading using a real testbed and these workload traces.

4.1 Experimental Setup

The experiments were all run using a testbed consisting of four standard HP servers, each with a dual-core Intel Xeon processor and an HP SmartArray 6400 controller connected to a rack-mounted disk enclosure with a SCSI backplane. All the servers were running Windows Server 2003 SP2. For the purposes of trace replay, data volumes were accessed as raw block devices rather than as file systems, since our traces are at the block level.

The disk enclosures were populated with 56 Seagate Cheetah 15,000 RPM disks: 28 of size 36 GB and 28 of size 146 GB. The servers were connected via a switched 1 Gbps Ethernet. The device driver for the SmartArray 6400 does not support physical spin-down and spin-up of the disks, highlighting the fact that disk spin-down is not standard practice in enterprise storage systems today. We did not have access to the driver source code and hence were forced to emulate the power state of each volume in a software layer. This layer delays requests on a spun-down volume until the volume is spun up; it also models the power consumed by each volume based on the number and type of disks and the emulated spin state.

The parameters for emulating the power state of the disks used in the testbed are shown in Table 3. These parameters were derived manually from the voltage/current profiles given in the Seagate Cheetah 15K.4 SCSI product manual [26] (Section 6, Figures 4 and 5). The steady-state power consumption when spun up is based on the current draw of both the 12 V input line (which powers the motor) and the 5 V line (which powers the electronics); the power consumption when spun down is based on the 5 V current only. The energy cost of spinning up is defined as the difference between the total energy used while the disk was spinning up, and that used if the disk were idle and spinning for that duration. We do not model the energy cost of doing I/O over and above that of keeping the disk electronics powered and the platter spinning; in general, this is difficult to model for an arbitrary workload and is also relatively small.

To drive the experiments we used real-time replay of the data center traces that we analyzed in Section 2. Using all the trace data would have required one week per experimental run. To make this tractable, each trace was split into seven one-day (24-hour) traces. These traces were statically analyzed to find the “least idle” and the

Time to spin up (36 GB disk)	10 s
Time to spin up (146 GB disk)	15 s
Energy cost of spinning up	20 J
Power when spun up	12 W
Power when spun down	2.6 W

Table 3: Energy parameters for Seagate Cheetah 15K.4

Rack	Server	Function	#volumes
1	usr	User files	3
	mds	Media server	2
	prn	Print server	2
	hm	H/w monitoring	2
2	src2	Source control	3
	proj	Project files	5
	wdev	Test web server	4
3	rsrch	Research projects	3
	prxy	Firewall/web proxy	1
	src1	Source control	2
	stg	Web staging	2
	ts	Terminal server	1
	web	Web/SQL server	4

Table 4: Servers grouped by rack

“most idle” days. Averaged across all volumes, the least idle day provides the smallest potential amount of idle time for write off-loading, whereas the most idle day provides the largest. The least idle day ran from midnight on Monday 26th February 2007 to midnight on the following day; it had 35 million requests with 73% reads. The most idle day ran from midnight on Sunday 25th February to midnight on the following day; it had 21 million requests with 70% reads. These two days represent the worst and the best case for energy savings using write off-loading, and hence our evaluation is based on them.

To emulate the traced data center volumes, sets of volumes were mapped on to the testbed. The entire testbed’s disk capacity is much smaller than the original traced servers. Therefore the traced servers were divided into three sets or “racks” (Table 4). Experiments were run for each rack independently; all the volumes in a single rack were simultaneously emulated on the testbed. Two of the 36 volumes (**prxy/1** and **src1/0**) could not be accommodated on the testbed with enough disks to sustain the offered load, so they were omitted. Due to the physical limitations of the testbed, the mapping does not keep volumes from the same original server on the same testbed server, or vice versa. However, our results show that the peak network load imposed by write off-loading is less than 7% of the network capacity; hence remapping does not significantly affect the system’s performance. Each volume was mapped to a RAID-1 or RAID-5 array with sufficient capacity to replay the volume trace.

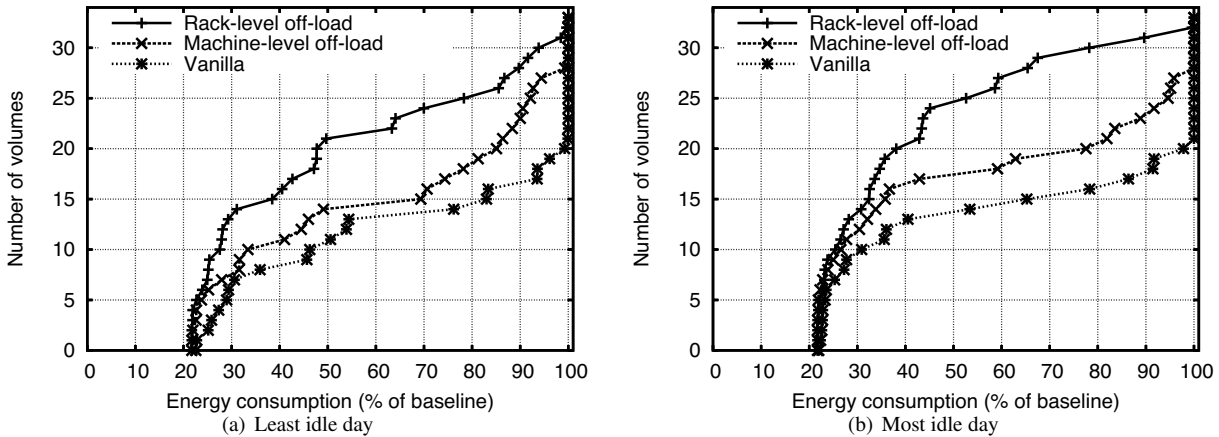


Figure 6: CDF of energy consumed as a percentage of baseline

A per-server trace replay component was used to replay the traces and gather performance metrics. The start of trace replay was synchronized across all servers. Each trace event was then converted to an I/O request sent to the corresponding emulated volume at the time specified by the timestamp: i.e., the traces were replayed “open-loop” in real time. This is necessary because the block-level traces do not capture higher-level dependencies between requests. However, requests which accessed overlapping block ranges were serialized, under the assumption that such requests would not be issued concurrently.

When configured for write off-loading each emulated volume was assigned both a manager and a logger. The logger used a 4 GB partition at the end of the volume; on hardware with flash or other solid-state devices the logger would run on the solid-state component instead. All manager and logger components on each server were linked together into a single user-level process along with the trace replay component. This component opened each server volume as a raw Windows block device; trace I/Os were then converted into read and write requests on these devices. Communication between managers and loggers is in-process if on the same physical server; otherwise we use UDP for broadcast and TCP for unicast.

In the experiments we evaluated four configurations:

- *baseline*: Volumes are never spun down. This gives no energy savings and no performance overhead.
- *vanilla*: Volumes spin down when idle, and spin up again on the next request, whether read or write.
- *machine-level off-load*: Write off-loading is enabled but managers can only off-load writes to loggers running on the same server: here the “server” is the original traced server, not the testbed replay server.
- *rack-level off-load*: Managers can off-load writes to any logger in the rack.

The configurations which place volumes in standby require an idle period before initiating standby. In the

vanilla configuration we use an idle period of 60 seconds. For the two off-load configurations, a volume is placed in standby after 60 seconds of no reads and 10 seconds of no writes. Each off-load manager also limits the amount of off-loaded data to 1 GB: on reaching this limit, the manager spins up the volume in the background.

In the remainder of this section we present the summarized results of our experimental runs. Each result is presented both for the “most idle” day and the “least idle” day, and for each of the four configurations. For a given day and configuration, results are aggregated across racks; although the experiments were run sequentially, this emulates all three racks running concurrently with off-loading (if enabled) happening at the rack or machine level depending on the configuration.

4.2 Energy Savings

Figures 6(a) and 6(b) show the CDFs of energy consumed per volume for the least idle day and most idle day, respectively. Obviously, in the baseline case, all disks would always be spun up, and volumes would always be at their maximum power level. Hence we normalize the power consumption of each volume in the other three configurations by the corresponding baseline value. All three configurations on both days save significant amounts of energy compared to the baseline, as seen by the number of volumes that use 50% or lower energy compared to baseline. Figure 7 summarizes these results showing the mean and peak power consumption across all volumes, again as a percentage of the baseline value.

For the least idle day, of the three non-baseline configurations, the vanilla configuration consumes the most energy: 72% of baseline. This is because it does not utilize write off-loading to lengthen the idle periods. Machine-level off-loading is able to do this, and hence uses less energy: 64% of the baseline. However, the energy savings

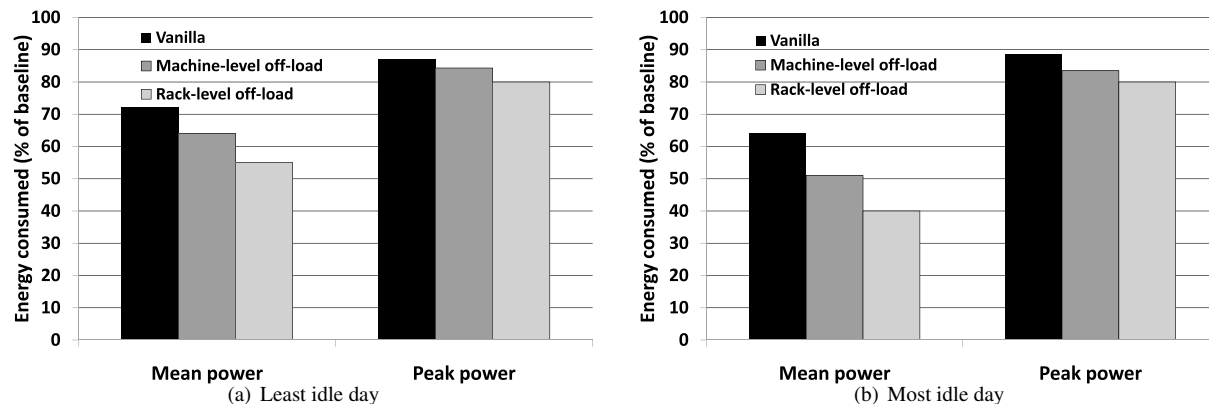


Figure 7: Total power consumption as percentage of baseline

are limited by the need to keep one volume spinning per machine to absorb the off-loaded writes. This means that at all times at least 13 of 34 volumes were kept spinning. Rack-level off-loading uses the least energy of all (55% of baseline) since it does not suffer from the limitations of the other two approaches. With rack-level off-loading, potentially a single spun-up volume could absorb the off-loads for the entire rack.

For the most idle day, all three non-baseline configurations improve their energy savings by exploiting the increased amount of idleness. As on the least idle day, vanilla uses significantly less energy (64%) than baseline; machine-level off-load does better (51%) than vanilla; and rack-level off-load does even better (40%).

The peak power results in Figure 7 show that vanilla reduces peak power to 87–89% of the baseline; machine-level off-load to 83–84%, and rack-level off-load to 80%. Unlike the mean power usage these results show that there is not a significant difference between the most and least idle days. This is because the power usage varies considerably over the time scale of minutes, as shown in Figure 3, and hence even on a day with a low mean power usage, the peak could be quite high. However, there is still a significant difference between the off-loading and non-off-loading configurations.

4.3 Performance Impact

We now evaluate the performance impact of spinning disks down, both with and without off-loading. We measured the response time of each read and write request in each configuration, on each of the test days. Figure 8 shows the response time distributions for reads and writes, for the two days, aggregated over all volumes. Since most requests have the same response time in all configurations, we put the y -axis on a log scale to highlight the differences between the configurations. The x -axis is also on a log scale, since response times vary from

tens of milliseconds in the common case to over 15 seconds in the worst case. For example, the baseline curve in Figure 8(a) shows that 0.01 (1%) of the read requests in the baseline configuration on the least idle day had a response time of more than 100 ms.

We see that for the majority of requests, the performance was identical in all configurations. For a small fraction of the requests, we see a long tail in some cases, going out to over 10 seconds. This tail represents requests that needed to wait for a disk to spin up. In the vanilla configuration both reads and writes are impacted by this effect. For the machine-level and rack-level off-load only reads are affected: for write requests they track baseline performance up to and beyond the 0.0001 point, i.e., for 99.99% or more of the requests. For a very small number of requests (fewer than 0.01%) on the least idle day, the machine-level off-load does worse than the baseline or rack-level off-load; this is because in the case of a heavy burst of write requests, the machine-level off-load cannot spread the load across loggers on multiple servers, whereas the rack-level off-load can do this.

These results confirm our expectation that spinning disks down causes a large penalty for a small number of requests. This penalty occurs for both reads and writes if we do not off-load writes, and only for reads if we do. It is worth noting that both the length and thickness of this tail are increased by an artifact of our experimental setup. Since we replay the traces open-loop, all requests that arrive while a disk is spinning up will be queued, and simultaneously released to the disk when it spins up. Since the spin-up period is relatively long (10–15 seconds), a large number of requests could arrive during this period, which cause an additional large queuing delay even after the disk is spun up. For example, the tail for the vanilla configuration in Figure 8(c) goes out to 56 seconds: this is due to a single episode in which 5,000 requests were queued while waiting for a volume to spin up, and 22 of these requests suffered extremely high delays as a result.

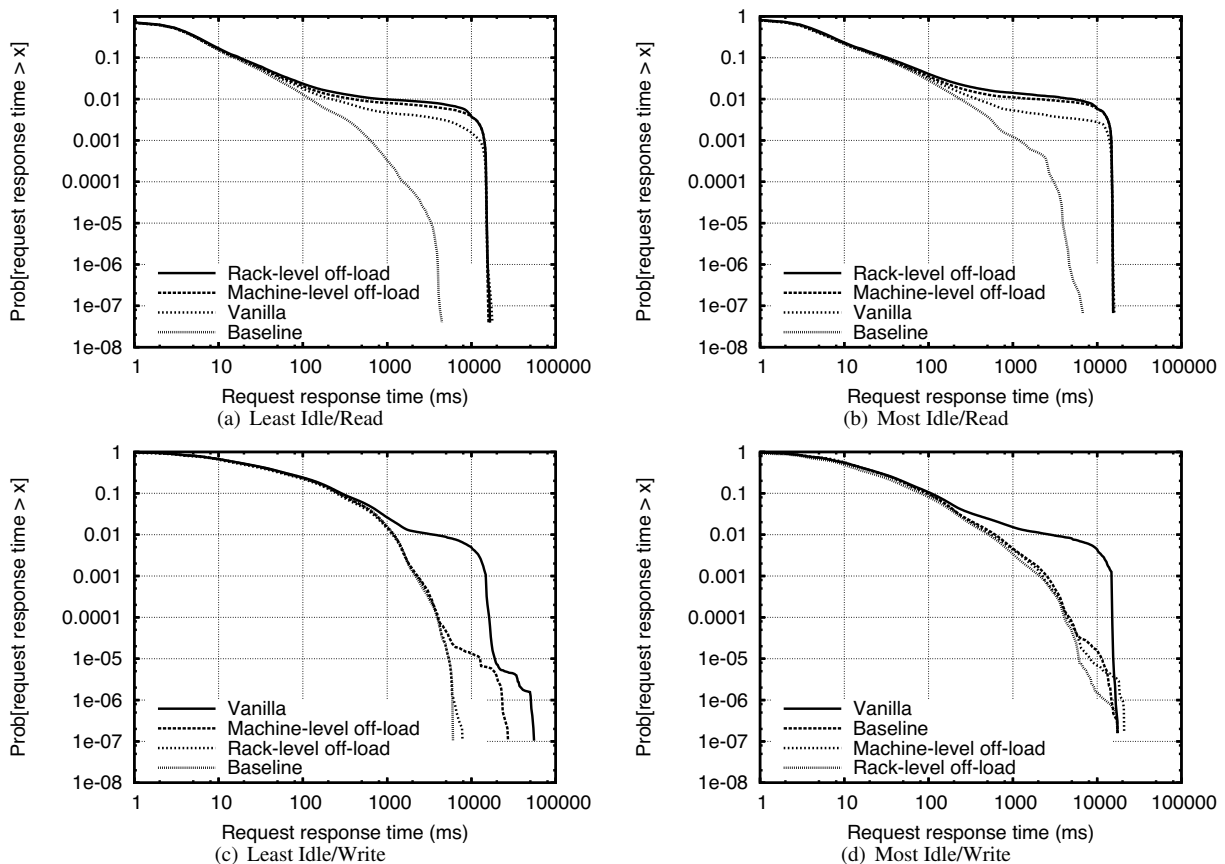


Figure 8: Response time distributions: the x -axis shows response times on a log scale, and the y -axis shows the fraction (also on a log scale) of requests with response times higher than some value

In reality, many of the requests in a single such burst would have been issued by the application in a closed loop, i.e., serialized one after another. Hence, delaying the first request would have prevented additional requests from being issued, avoiding the large queues and queuing delay. Further, if the request burst was created by a non-interactive application, for example a backup, then an initial delay of 10–15 seconds is acceptable. For interactive applications, of course, this first-byte delay will be visible to the user; if a volume supports interactive applications that cannot tolerate this delay even infrequently, then write off-loading or indeed any kind of spin-down should not be enabled for that volume.

We now present some summary statistics on the response time distributions. Figure 9(a) shows the median response time; as expected, there is no penalty for spinning disks down or off-loading. Figure 9(b) shows the mean response time. Here we see the effect of the skewed response time distribution, i.e., the long thin tail, which causes a small number of requests to increase the mean significantly. For reads, all the non-baseline configurations have a high mean response time. The off-

load configurations do worse than vanilla, because they spin down more often (and save more energy): hence a burst of read requests is more likely to hit a spun-down volume in these cases. For the same reason the rack-level off-load has a higher mean response time than the machine-level off-load.

In the case of writes, the mean is high for vanilla, but the off-load configurations do slightly better than the baseline case, with the rack-level off-load having the best performance of all. This is because logger writes have good locality, since they use a log structure, even if the original access pattern does not have good locality. Further, during bursts of high write load, rack-level off-load is able to load-balance across multiple loggers on multiple servers, and hence delivers the best performance.

Figure 10 shows the percentage of requests incurring a spin-up delay in each case. Obviously, the baseline configuration has no spin-up delays, and the off-load configurations do not have spin-up delays on write requests. Reads for rack-level/machine-level off-load, and all requests for vanilla, have significant (up to 1.2%) numbers of spin-up delays, which skews the mean response time

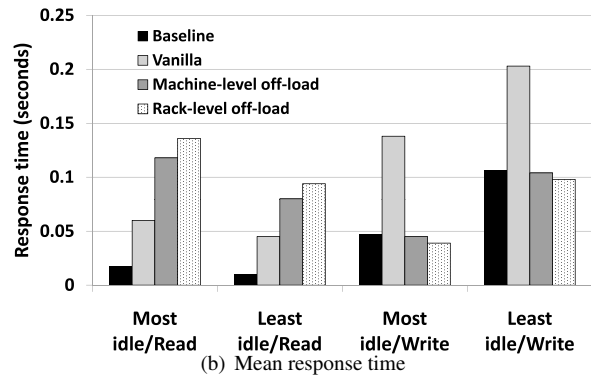
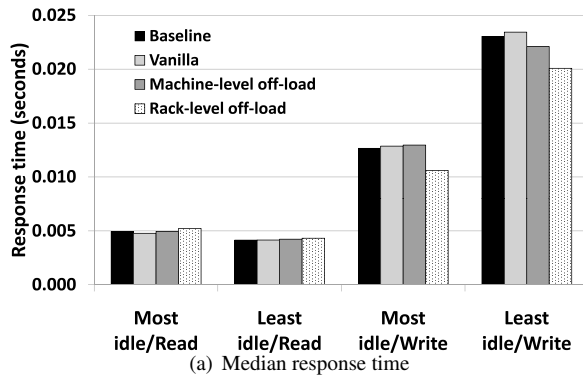


Figure 9: Median and mean response times

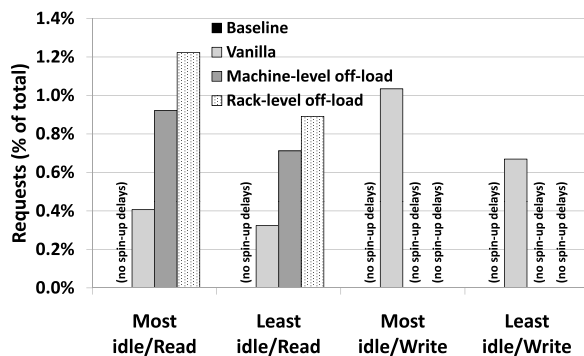


Figure 10: Percentage of requests incurring spin-up delays

for these cases. However, as remarked previously, some of this effect is an artifact of open-loop replay.

Figure 11(a) shows the 95th percentile response times respectively for the different cases. There are only minor differences between the different configurations, since much fewer than 5% of requests see spin-up delays in any configurations. Finally, Figure 11(b) shows the maximum response time for each case. For reads, the non-baseline configurations have similar worst-case performance to each other, although worse than the baseline: this is due to spin-up delays. For writes, all configurations have similar performance on the most idle day. On the least idle day, there is a large penalty for vanilla due to a combination of spin-up delay and queuing delay for a large write burst as previously discussed. Machine-level off-load also has a significant though lower penalty, due to the lack of load balancing on a heavy write burst. Rack-level off-load is able to load-balance such bursts and hence has a much better worst-case performance.

In summary, all configurations have comparable performance to the baseline case for a majority of requests; however, reads in the off-load configurations and both reads and writes in the vanilla configuration have a long

thin tail, which is unavoidable. Finally, rack-level off-load consistently outperforms machine-level off-load on both energy savings and write performance, but has worse read performance and adds network traffic to the system. Administrators can configure the logger views on a per-manager basis to provide the appropriate trade-off between these metrics.

4.4 Network Usage

We also measured the network overheads of rack-level off-loading across both test days combined. These are summarized in Table 5. Note that the bandwidth usage is based on communications between managers and loggers that belong to different servers in the original trace. In other words, this measures the network overheads if write off-loading had been run on the original traced servers rather than the testbed servers. Thus there are no network overheads for machine-level off-load and of course none for the baseline or vanilla configurations.

The average bandwidth usage is low compared to the bandwidth available in a typical data center. The peak bandwidth usage can easily be supported by gigabit networks, which are widespread in enterprise data centers. The mean RPC round-trip latency is the amount of additional latency incurred by requests due to manager-logger communication.

The last two entries in Table 5 show that a substantial fraction of write requests were off-loaded, but only a very small fraction of reads were remote. A remote read is one that required the manager to interact with a logger because some or all of the requested blocks had been off-loaded. This is the expected behavior: due to main-memory buffer caches, very few recently-written blocks are read back from the storage layer. In other words, most off-loaded blocks will be overwritten or reclaimed before they are read again. The small fraction of remote reads also justifies our decision to optimize the loggers for write rather than read requests. Machine-

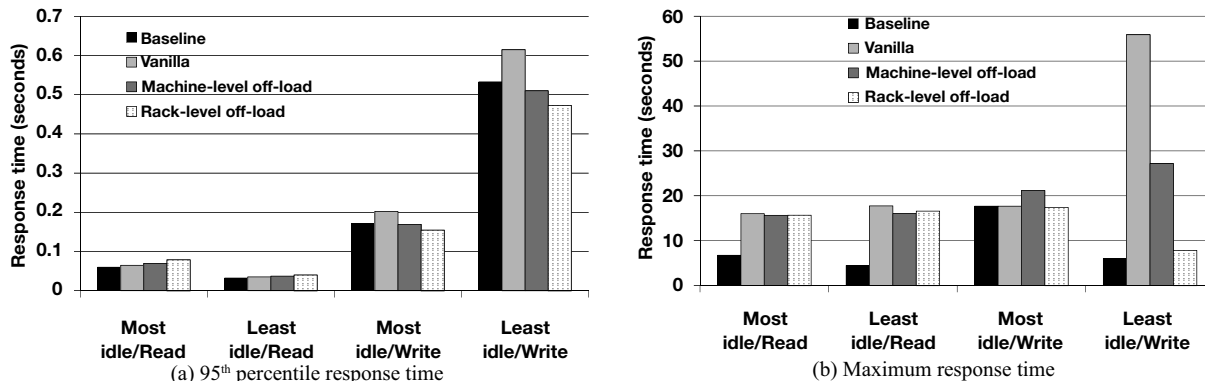


Figure 11: 95th percentile and maximum (worst-case) response times

Total network transfers over 2 days	46.5 GB
Average network bandwidth usage	2.31 Mbps
Peak bandwidth usage over 60 s	68.8 Mbps
Mean RPC round-trip latency	2.6 ms
Fraction of writes off-loaded	27%
Fraction of remote reads	3.8%

Table 5: Network overheads for rack-level off-loading

level off-loading gives similar results to rack-level off-loading, with 8.3% of writes off-loaded but only 0.78% of reads being remote.

5 Related Work

There has been considerable research in power management for laptop and mobile device storage [9, 18, 30] and also on high-level power management for data centers as a whole [6]. We focus on related work in power management for enterprise storage systems.

The closest related work is Massive Arrays of Idle Disks (MAID) [7] which has been proposed for replacing tape libraries as tertiary storage environments for very large-scale storage. MAIDs are storage components composed of 1,000s of disks holding hundreds of terabytes of storage. A subset of the disks are kept spinning, acting as a cache, while the rest are spun down. For standard RAID-based enterprise primary storage, this requires a minimum of two additional disks per volume. For non-archival storage this is an unacceptable overhead in terms of both energy and cost. In contrast, write off-loading does not require additional dedicated disks per volume or new hardware: we can opportunistically use any unused storage in the data center to store blocks.

Power-aware cache management [33] optimizes cache replacement for idle times rather than miss ratios, and shows power savings for OLTP, *cello* [22], and synthetic traces. This is orthogonal to our approach: any increase

in the inter-read time will result in increased energy savings with write off-loading. However, we observe that the enterprise workloads that we traced already have large inter-read times that we exploit using write off-loading.

DRPM [12] and Hibernator [32] are recently proposed approaches to save energy by using multi-speed disks (standard enterprise disks spin at a fixed rate of 10,000 or 15,000 rpm). They propose using lower spin speeds when load is low, which decreases power consumption while increasing access latency. However, multi-speed disks are not widely deployed today in the enterprise, and we do not believe their use is likely to become widespread in the near future.

Several approaches [16, 28, 29] have proposed power management schemes for RAID arrays at the RAID controller level or lower, which are orthogonal to write off-loading which works above the block device level. In particular, PARAID [28] uses different numbers of disks in the array for different load levels, and provides good power savings for read-mostly workloads. Hence it is complementary to write off-loading, which can convert a mixed workload to a read-only one.

Popular Data Concentration (PDC) [19] migrates data between disks to concentrate hot data on a few disks, allowing the remainder to spin down. This could be done within the disks on a volume: in this case it would be transparent and orthogonal to volume-level off-loading. PDC could potentially be done across volumes as well: however, this is not practical for most enterprise solutions since volumes will no longer be isolated from each other for performance, cannot be tuned individually, and acquire long-term data dependencies on each other.

Serverless file systems, such as xFS [1], attempt to distribute block storage and management across multiple networked machines, and use co-operative caching [8] to improve performance. By contrast, write off-loading is designed to save energy rather than reduce latency or in-

crease throughput. It also works at the block level, and rather than storing data remotely for days or weeks, a relatively small number of blocks are temporarily hosted on remote machines.

The log structure used to store off-loaded data and meta data is similar to those used in log-structured file systems [11, 21, 27]. However, log-structured file systems store all written data for the long term, whereas our logs store only off-loaded data, and temporarily.

Finally, various approaches to storage workload tracing and trace replay have been proposed in the research literature [2, 10, 15, 31]. We decided to use ETW for tracing since it is already supported on our traced servers, and it provides the functionality we needed (tracing block-level I/O requests) with low overhead.

6 Discussion

Hardware trends. Recently, there has been considerable interest in solid-state drives (SSD) for mobile devices and laptops [23, 24]. These drives currently vary in size from 4–32 GB, and use less power than disks. While SSD-based storage is likely to become widely used in laptops over the next 2–3 years, it is unlikely to replace disks in the enterprise in the foreseeable future due to the high per-GB costs and performance characteristics.

However, it is likely that solid-state memory (flash) will become common, either in hybrid drives or as a small generic block-level storage device on motherboards. Hybrid drives include a small amount of flash within the disk. This allows the drive to spin the physical disk down and use the flash as a persistent buffer cache. This is very similar to the idea of using battery-backed NVRAM as a buffer cache to reduce disk traffic [3].

Thus, if and when enterprise storage becomes fully SSD-based, write off-loading will offer few advantages. However, for the next decade or so we expect that server systems will continue to have disk-based systems, increasingly augmented with solid-state memory: by running loggers on the solid-state devices and using them for write off-loading, the power savings of write off-loading can be further increased.

Traditionally, spinning a disk up and down is viewed as increasing the stress on the drive and reducing the mean time to failure (MTTF). For the state-of-the-art enterprise class Seagate Cheetah 15K.4 SCSI drives, the MTTF calculations assume 200 power cycles per year. Recent research has re-examined some of the assumptions about factors that impact disk lifetime [20, 25] but has not examined the effect of spinning disks down: we see it as an open question what impact spinning up and down will have on enterprise disks.

Configuration and management. Write off-loading requires some level of administrator configuration and management. For example, an administrator might wish to disable write off-loading for some period of time on some set of disks, say a RAID array, hosting both managers and loggers. When this is desired, all data on loggers hosted on those disks must be reclaimed by their home volumes. Similarly, all data off-loaded by managers on those disks must be reclaimed and invalidated on the loggers that were storing them. This would be the procedure, for example, for decommissioning a volume that currently has write off-loading enabled.

System boot volumes typically should not have an off-load manager enabled (although they can certainly support a logger). This avoids off-loading blocks that are required for the system to boot.

7 Conclusion

In this paper we propose a technique called write off-loading to save energy in enterprise storage. It allows blocks written to one volume to be temporarily redirected to persistent storage elsewhere in an enterprise data center. This alters the I/O access pattern to the volume, generating significant idle periods during which the volume's disks can be spun down, thereby saving energy.

We analyzed the potential savings using real-world traces gathered for a week from the 13 servers in our building's data center. Our analysis shows that simply spinning disks down when idle saves significant energy. Further, write off-loading enables potentially much larger savings by creating longer idle periods. To validate the analysis we implemented write off-loading and measured its performance on a hardware testbed. The evaluation confirms the analysis results: just spinning disks down when idle reduces their energy consumption by 28–36%, and write off-loading increases the savings to 45–60%.

We believe that write off-loading is a viable technique for saving energy in enterprise storage. In order to use write off-loading, a system administrator needs to manage the trade-off between energy and performance. We are designing tools to help administrators decide how to save the most energy with the least performance impact.

Acknowledgements

We would like to thank our IT support staff at Microsoft Research Cambridge, particularly Nathan Jones for helping us to trace our data center servers. We would also like to thank our anonymous reviewers and our shepherd Erez Zadok.

References

- [1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, CO, Dec. 1995.
- [2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A file system to trace them all. In *Proc. USENIX Conference on File and Storage Technologies (FAST'04)*, San Francisco, CA, Mar.–Apr. 2004.
- [3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, Boston, MA, Oct. 1992.
- [4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'91)*, Pacific Grove, CA, Oct. 1991.
- [5] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proc. ACM International Conference on Supercomputing (ICS'03)*, San Francisco, CA, June 2003.
- [6] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'01)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.
- [7] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proc. ACM/IEEE Conference on Supercomputing*, Baltimore, MD, Nov. 2002.
- [8] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI'94)*, Monterey, CA, Nov. 1994.
- [9] F. Douglass, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proc. USENIX Winter 1994 Technical Conference*, San Francisco, CA, Jan. 1994.
- [10] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. Passive NFS tracing of email and research workloads. In *Proc. USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, Mar. 2003.
- [11] L. Ganesh, H. Weatherspoon, M. Balakrishnan, and K. Birman. Optimizing power consumption in large scale storage systems. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS'07)*, San Diego, CA, May 2007.
- [12] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proc. International Symposium on Computer Architecture (ISCA'03)*, San Diego, CA, June 2003.
- [13] S. Gurumurthi, J. Zhang, A. Sivasubramaniam, M. Kandemir, H. Franke, N. Vijaykrishnan, and M. Irwin. Interplay of energy and performance for disk arrays running transaction processing workloads. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS'03)*, Austin, TX, Mar. 2003.
- [14] Intel Corporation. *Dual-Core Intel® Xeon® Processor 5100 Series Datasheet*, Nov. 2006. Reference Number: 313355-002.
- [15] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *Proc. USENIX Conference on File and Storage Technologies (FAST'05)*, San Francisco, CA, Dec. 2005.
- [16] D. Li and J. Wang. EERAID: Energy efficient redundant and inexpensive disk arrays. In *Proc. 11th ACM SIGOPS European Workshop (SIGOPS-EW'04)*, Leuven, Belgium, Sept. 2004.
- [17] Microsoft. Event tracing. <http://msdn.microsoft.com/library/>, 2002. Platform SDK: Performance Monitoring, Event Tracing.
- [18] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the Blue file system. In *Proc. Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, Dec. 2004.
- [19] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *Proc. Annual International Conference on Supercomputing (ICS'04)*, June 2004.
- [20] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. USENIX Conference on File and Storage Technologies (FAST'07)*, San Jose, CA, Feb. 2007.

- [21] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'91)*, Pacific Grove, CA, Oct. 1991.
- [22] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proc. USENIX Winter 1993 Technical Conference*, San Diego, CA, Jan. 1993.
- [23] Samsung. NAND flash-based solid state disk product data sheet, Jan. 2007.
- [24] SanDisk. SSD UATA 5000 1.8" data sheet. Document No. 80-11-00001, Feb. 2007.
- [25] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proc. USENIX Conference on File and Storage Technologies (FAST'07)*, San Jose, CA, Feb. 2007.
- [26] Seagate Technology LLC, 920 Disc Drive, Scotts Valley, CA 95066-4544, USA. *Cheetah 15K.4 SCSI Product Manual*, Rev. D edition, May 2005. Publication number: 100220456.
- [27] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proc. USENIX Winter 1993 Conference*, San Diego, CA, Jan. 1993.
- [28] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning. PARAID: The gear-shifting power-aware RAID. In *Proc. USENIX Conference on File and Storage Technologies (FAST'07)*, San Jose, CA, Feb. 2007.
- [29] X. Yao and J. Wang. Rimac: A novel redundancy-based hierarchical cache architecture for energy efficient, high performance storage systems. In *Proc. EuroSys Conference*, Leuven, Belgium, Apr. 2006.
- [30] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proc. USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, Mar. 2003.
- [31] N. Zhu, J. Chen, and T. Chiueh. TBBT: Scalable and accurate trace replay for file server evaluation. In *Proc. USENIX Conference on File and Storage Technologies (FAST'05)*, San Francisco, CA, Dec. 2005.
- [32] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'05)*, Brighton, United Kingdom, Oct. 2005.
- [33] Q. Zhu and Y. Zhou. Power-aware storage cache management. *IEEE Trans. Computers*, 54(5):587–602, 2005.

Avoiding the Disk Bottleneck in the Data Domain Deduplication File System

Benjamin Zhu
Data Domain, Inc.

Kai Li
Data Domain, Inc. and Princeton University

Hugo Patterson
Data Domain, Inc.

Abstract

Disk-based deduplication storage has emerged as the new-generation storage system for enterprise data protection to replace tape libraries. Deduplication removes redundant data segments to compress data into a highly compact form and makes it economical to store backups on disk instead of tape. A crucial requirement for enterprise data protection is high throughput, typically over 100 MB/sec, which enables backups to complete quickly. A significant challenge is to identify and eliminate duplicate data segments at this rate on a low-cost system that cannot afford enough RAM to store an index of the stored segments and may be forced to access an on-disk index for every input segment.

This paper describes three techniques employed in the production Data Domain deduplication file system to relieve the disk bottleneck. These techniques include: (1) the Summary Vector, a compact in-memory data structure for identifying new segments; (2) Stream-Informed Segment Layout, a data layout method to improve on-disk locality for sequentially accessed segments; and (3) Locality Preserved Caching, which maintains the locality of the fingerprints of duplicate segments to achieve high cache hit ratios. Together, they can remove 99% of the disk accesses for deduplication of real world workloads. These techniques enable a modern two-socket dual-core system to run at 90% CPU utilization with only one shelf of 15 disks and achieve 100 MB/sec for single-stream throughput and 210 MB/sec for multi-stream throughput.

1 Introduction

The massive storage requirements for data protection have presented a serious problem for data centers. Typically, data centers perform a weekly full backup of all the data on their primary storage systems to secondary storage devices where they keep these backups for weeks to months. In addition, they may perform daily incremental backups that copy only the data which has changed since the last backup. The frequency, type and retention of backups vary for different kinds of data, but it is common for the secondary storage to hold 10 to 20 times more data than the primary storage. For disaster recovery, additional offsite copies may double the secondary storage capacity needed. If the data is transferred offsite over a wide area network, the network bandwidth requirement can be enormous.

Given the data protection use case, there are two main requirements for a secondary storage system storing backup data. The first is low cost so that storing backups and moving copies offsite does not end up costing significantly more than storing the primary data. The second is high performance so that backups can complete in a timely fashion. In many cases, backups must complete overnight so the load of performing backups does not interfere with normal daytime usage.

The traditional solution has been to use tape libraries as secondary storage devices and to transfer physical tapes for disaster recovery. Tape cartridges cost a small fraction of disk storage systems and they have good sequential transfer rates in the neighborhood of 100 MB/sec. But, managing cartridges is a manual process that is expensive and error prone. It is quite common for restores to fail because a tape cartridge cannot be located or has been damaged during handling. Further, random access performance, needed for data restores, is extremely poor. Disk-based storage systems and network replication would be much preferred if they were affordable.

During the past few years, disk-based, “deduplication” storage systems have been introduced for data protection [QD02, MCM01, KDLT04, Dat05, JDT05]. Such systems compress data by removing duplicate data across files and often across all the data in a storage system. Some implementations achieve a 20:1 compression ratio (total data size divided by physical space used) for 3 months of backup data using the daily-incremental and weekly-full backup policy. By substantially reducing the footprint of versioned data, deduplication can make the costs of storage on disk and tape comparable and make replicating data over a WAN to a remote site for disaster recovery practical.

The specific deduplication approach varies among system vendors. Certainly the different approaches vary in how effectively they reduce data. But, the goal of this paper is not to investigate how to get the greatest data reduction, but rather how to do deduplication at high speed in order to meet the performance requirement for secondary storage used for data protection.

The most widely used deduplication method for secondary storage, which we call Identical Segment Deduplication, breaks a data file or stream into contiguous segments and eliminates duplicate copies of identical segments. Several emerging commercial systems have used this approach.

The focus of this paper is to show how to implement a high-throughput Identical Segment Deduplication storage system at low system cost. The key performance challenge is finding duplicate segments. Given a segment size of 8 KB and a performance target of 100 MB/sec, a deduplication system must process approximately 12,000 segments per second.

An in-memory index of all segment fingerprints could easily achieve this performance, but the size of the index would limit system size and increase system cost. Consider a segment size of 8 KB and a segment fingerprint size of 20 bytes. Supporting 8 TB worth of unique segments, would require 20 GB just to store the fingerprints.

An alternative approach is to maintain an on-disk index of segment fingerprints and use a cache to accelerate segment index accesses. Unfortunately, a traditional cache would not be effective for this workload. Since fingerprint values are random, there is no spatial locality in the segment index accesses. Moreover, because the backup workload streams large data sets through the system, there is very little temporal locality. Most segments are referenced just once every week during the full backup of one particular system. Reference-based caching algorithms such as LRU do not work well for such workloads. The Venti system, for example, implemented such a cache [QD02]. Its combination of index and block caches only improves its write throughput by about 16% (from 5.6MB/sec to 6.5MB/sec) even with 8 parallel disk index lookups. The primary reason is due to its low cache hit ratios.

With low cache hit ratios, most index lookups require disk operations. If each index lookup requires a disk access which may take 10 msec and 8 disks are used for index lookups in parallel, the write throughput will be about 6.4MB/sec, roughly corresponding to Venti's throughput of less than 6.5MB/sec with 8 drives. While Venti's performance may be adequate for the archival usage of a small workgroup, it's a far cry from the throughput goal of deduplicating at 100 MB/sec to

compete with high-end tape libraries. Achieving 100 MB/sec, would require 125 disks doing index lookups in parallel! This would increase the system cost of deduplication storage to an unattainable level.

Our key idea is to use a combination of three methods to reduce the need for on-disk index lookups during the deduplication process. We present in detail each of the three techniques used in the production Data Domain deduplication file system. The first is to use a Bloom filter, which we call a *Summary Vector*, as the summary data structure to test if a data segment is new to the system. It avoids wasted lookups for segments that do not exist in the index. The second is to store data segments and their fingerprints in the same order that they occur in a data file or stream. Such *Stream-Informed Segment Layout* (SISL) creates spatial locality for segment and fingerprint accesses. The third, called *Locality Preserved Caching*, takes advantage of the segment layout to fetch and cache groups of segment fingerprints that are likely to be accessed together. A single disk access can result in many cache hits and thus avoid many on-disk index lookups.

Our evaluation shows that these techniques are effective in removing the disk bottleneck in an Identical Segment Deduplication storage system. For a system running on a server with two dual-core CPUs with one shelf of 15 drives, these techniques can eliminate about 99% of index lookups for variable-length segments with an average size of about 8 KB. We show that the system indeed delivers high throughput: achieving over 100 MB/sec for single-stream write and read performance, and over 210 MB/sec for multi-stream write performance. This is an order-of-magnitude throughput improvement over the parallel indexing techniques presented in the Venti system.

The rest of the paper is organized as follows. Section 2 presents challenges and observations in designing a deduplication storage system for data protection. Section 3 describes the software architecture of the production Data Domain deduplication file system. Section 4 presents our methods for avoiding the disk bottleneck. Section 5 shows our experimental results. Section 6 gives an overview of the related work, and Section 7 draws conclusions.

2 Challenges and Observations

2.1 Variable vs. Fixed Length Segments

An Identical Segment Deduplication system could choose to use either fixed length segments or variable length segments created in a content dependent manner. Fixed length segments are the same as the fixed-size blocks of many non-deduplication file systems. For the purposes of this discussion, extents that are multiples of

some underlying fixed size unit such as a disk sector are the same as fixed-size blocks.

Variable-length segments can be any number of bytes in length within some range. They are the result of partitioning a file or data stream in a content dependent manner [Man93, BDH94].

The main advantage of a fixed segment size is simplicity. A conventional file system can create fixed-size blocks in the usual way and a deduplication process can then be applied to deduplicate those fixed-size blocks or segments. The approach is effective at deduplicating whole files that are identical because every block of identical files will of course be identical.

In backup applications, single files are backup images that are made up of large numbers of component files. These files are rarely entirely identical even when they are successive backups of the same file system. A single addition, deletion, or change of any component file can easily shift the remaining image content. Even if no other file has changed, the shift would cause each fixed sized segment to be different than it was last time, containing some bytes from one neighbor and giving up some bytes to its other neighbor. The approach of partitioning the data into variable length segments based on content allows a segment to grow or shrink as needed so the remaining segments can be identical to previously stored segments.

Even for storing individual files, variable length segments have an advantage. Many files are very similar to, but not identical to other versions of the same file. Variable length segments can accommodate these differences and maximize the number of identical segments.

Because variable length segments are essential for deduplication of the shifted content of backup images, we have chosen them over fixed-length segments.

2.2 Segment Size

Whether fixed or variable sized, the choice of average segment size is difficult because of its impact on compression and performance. The smaller the segments, the more duplicate segments there will be. Put another way, if there is a small modification to a file, the smaller the segment, the smaller the new data that must be stored and the more of the file's bytes will be in duplicate segments. Within limits, smaller segments will result in a better compression ratio.

On the other hand, with smaller segments, there are more segments to process which reduces performance. At a minimum, more segments mean more times through the deduplication loop, but it is also likely to mean more on-disk index lookups.

With smaller segments, there are more segments to manage. Since each segment requires the same metadata size, smaller segments will require more storage footprint for their metadata, and the segment fingerprints for fewer total user bytes can be cached in a given amount of memory. The segment index is larger. There are more updates to the index. To the extent that any data structures scale with the number of segments, they will limit the overall capacity of the system. Since commodity servers typically have a hard limit on the amount of physical memory in a system, the decision on the segment size can greatly affect the cost of the system.

A well-designed duplication storage system should have the smallest segment size possible given the throughput and capacity requirements for the product. After several iterative design processes, we have chosen to use 8 KB as the average segment size for the variable sized data segments in our deduplication storage system.

2.3 Performance-Capacity Balance

A secondary storage system used for data protection must support a reasonable balance between capacity and performance. Since backups must complete within a fixed backup window time, a system with a given performance can only backup so much data within the backup window. Further, given a fixed retention period for the data being backed up, the storage system needs only so much capacity to retain the backups that can complete within the backup window. Conversely, given a particular storage capacity, backup policy, and deduplication efficiency, it is possible to compute the throughput that the system must sustain to justify the capacity. This balance between performance and capacity motivates the need to achieve good system performance with only a small number of disk drives.

Assuming a backup policy of weekly fulls and daily incrementals with a retention period of 15 weeks and a system that achieves a 20x compression ratio storing backups for such a policy, as a rough rule of thumb, it requires approximately as much capacity as the primary data to store all the backup images. That is, for 1 TB of primary data, the deduplication secondary storage would consume approximately 1 TB of physical capacity to store the 15 weeks of backups.

Weekly full backups are commonly done over the weekend with a backup window of 16 hours. The balance of the weekend is reserved for restarting failed backups or making additional copies. Using the rule of thumb above, 1 TB of capacity can protect approximately 1 TB of primary data. All of that must be backed up within the 16-hour backup window which implies a throughput of about 18 MB/sec per terabyte of capacity.

Following this logic, a system with a shelf of 15 SATA drives each with a capacity of 500 GB and a total usable capacity after RAID, spares, and other overhead of 6 TB could protect 6 TB of primary storage and must therefore be able to sustain over 100 MB/sec of deduplication throughput.

2.4 Fingerprint vs. Byte Comparisons

An Identical Segment Deduplication storage system needs a method to determine that two segments are identical. This could be done with a byte by byte comparison of the newly written segment with the previously stored segment. However, such a comparison is only possible by first reading the previously stored segment from disk. This would be much more onerous than looking up a segment in an index and would make it extremely difficult if not impossible to maintain the needed throughput.

To avoid this overhead, we rely on comparisons of segment fingerprints to determine the identity of a segment. The fingerprint is a collision-resistant hash value computed over the content of each segment. SHA-1 is such a collision-resistant function [NIST95]. At a 160-bit output value, the probability of fingerprint collision by a pair of different segments is extremely small, many orders of magnitude smaller than hardware error rates [QD02]. When data corruption occurs, it will almost certainly be the result of undetected errors in RAM, IO busses, network transfers, disk storage devices, other hardware components or software errors and not from a collision.

3 Deduplication Storage System Architecture

To provide the context for presenting our methods for avoiding the disk bottleneck, this section describes the architecture of the production Data Domain File System, DDFS, for which Identical Segment Deduplication is an integral feature. Note that the methods presented in the next section are general and can apply to other Identical Segment Deduplication storage systems.

At the highest level, DDFS breaks a file into variable-length segments in a content dependent manner [Man93, BDH94] and computes a fingerprint for each segment. DDFS uses the fingerprints both to identify duplicate segments and as part of a segment descriptor used to reference a segment. It represents files as sequences of segment fingerprints. During writes, DDFS identifies duplicate segments and does its best to store only one copy of any particular segment. Before storing a new segment, DDFS uses a variation of the Ziv-Lempel algorithm to compress the segment [ZL77].

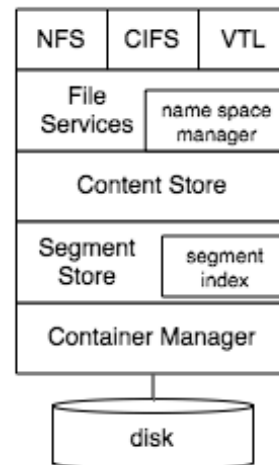


Figure 1: Data Domain File System architecture.

Figure 1 is a block diagram of DDFS, which is made up of a stack of software components. At the top of the stack, DDFS supports multiple access protocols which are layered on a common File Services interface. Supported protocols include NFS, CIFS, and a virtual tape library interface (VTL).

When a data stream enters the system, it goes through one of the standard interfaces to the generic File Services layer, which manages the name space and file metadata. The File Services layer forwards write requests to Content Store which manages the data content within a file. Content Store breaks a data stream into segments, uses Segment Store to perform deduplication, and keeps track of the references for a file. Segment Store does the actual work of deduplication. It packs deduplicated (unique) segments into relatively large units, compresses such units using a variation of Ziv-Lempel algorithm to further compress the data, and then writes the compressed results into containers supported by Container Manager.

To read a data stream from the system, a client drives the read operation through one of the standard interfaces and the File Services Layer. Content Store uses the references to deduplicated segments to deliver the desired data stream to the client. Segment Store prefetches, decompresses, reads and caches data segments from Container Manager.

The following describes the Content Store, Segment Store and the Container Manager in detail and discusses our design decisions.

3.1 Content Store

Content Store implements byte-range writes and reads for deduplicated data objects, where an object is a linear

sequence of client data bytes and has intrinsic and client-settable attributes or metadata. An object may be a conventional file, a backup image of an entire volume or a tape cartridge.

To write a range of bytes into an object, Content Store performs several operations.

- *Anchoring* partitions the byte range into variable-length segments in a content dependent manner [Man93, BDH94].
- *Segment fingerprinting* computes the SHA-1 hash and generates the segment descriptor based on it. Each segment descriptor contains per segment information of at least fingerprint and size
- *Segment mapping* builds the tree of segments that records the mapping between object byte ranges and segment descriptors. The goal is to represent a data object using references to deduplicated segments.

To read a range of bytes in an object, Content Store traverses the tree of segments created by the segment mapping operation above to obtain the segment descriptors for the relevant segments. It fetches the segments from Segment Store and returns the requested byte range to the client.

3.2 Segment Store

Segment Store is essentially a database of segments keyed by their segment descriptors. To support writes, it accepts segments with their segment descriptors and stores them. To support reads, it fetches segments designated by their segment descriptors.

To write a data segment, Segment Store performs several operations.

- *Segment filtering* determines if a segment is a duplicate. This is the key operation to deduplicate segments and may trigger disk I/Os, thus its overhead can significantly impact throughput performance.
- *Container packing* adds segments to be stored to a container which is the unit of storage in the system. The packing operation also compresses segment data using a variation of the Ziv-Lempel algorithm. A container, when fully packed, is appended to the Container Manager.
- *Segment Indexing* updates the segment index that maps segment descriptors to the container holding the segment, after the container has been appended to the Container Manager.

To read a data segment, Segment Store performs the following operations.

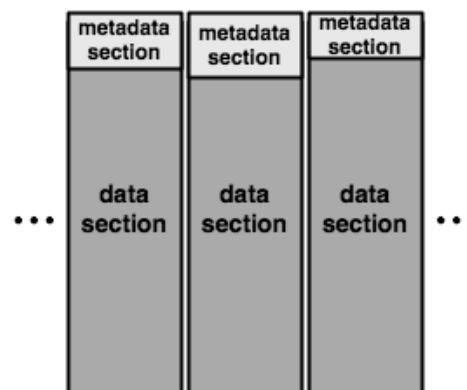


Figure 2: Containers are self-describing, immutable, units of storage several megabytes in size. All segments are stored in containers.

- *Segment lookup* finds the container storing the requested segment. This operation may trigger disk I/Os to look in the on-disk index, thus it is throughput sensitive.
- *Container retrieval* reads the relevant portion of the indicated container by invoking the Container Manager.
- *Container unpacking* decompresses the retrieved portion of the container and returns the requested data segment.

3.3 Container Manager

The Container Manager provides a storage container log abstraction, not a block abstraction, to Segment Store. Containers, shown in Figure 2, are self-describing in that a metadata section includes the segment descriptors for the stored segments. They are immutable in that new containers can be appended and old containers deleted, but containers cannot be modified once written. When Segment Store appends a container, the Container Manager returns a container ID which is unique over the life of the system.

The Container Manager is responsible for allocating, deallocating, reading, writing and reliably storing containers. It supports reads of the metadata section or a portion of the data section, but it only supports appends of whole containers. If a container is not full but needs to be written to disk, it is padded out to its full size.

Container Manager is built on top of standard block storage. Advanced techniques such as Software RAID-6, continuous data scrubbing, container verification, and end to end data checks are applied to ensure a high level of data integrity and reliability.

The container abstraction offers several benefits.

- The fixed container size makes container allocation and deallocation easy.
- The large granularity of a container write achieves high disk throughput utilization.
- A properly sized container size allows efficient full-stripe RAID writes, which enables an efficient software RAID implementation at the storage layer.

4 Acceleration Methods

This section presents three methods to accelerate the deduplication process in our deduplication storage system: summary vector, stream-informed data layout, and locality preserved caching. The combination of these methods allows our system to avoid about 99% of the disk I/Os required by a system relying on index lookups alone. The following describes each of the three techniques in detail.

4.1 Summary Vector

The purpose of the Summary Vector is to reduce the number of times that the system goes to disk to look for a duplicate segment only to find that none exists. One can think of the Summary Vector as an in-memory, conservative summary of the segment index. If the Summary Vector indicates a segment is not in the index, then there is no point in looking further for the segment; the segment is new and should be stored. On the other hand, being only an approximation of the index, if the Summary Vector indicates the segment is in the index, there is a high probability that the segment is actually in the segment index, but there is no guarantee.

The Summary Vector implements the following operations:

- `Init()`
- `Insert(fingerprint)`
- `Lookup(fingerprint)`

We use a Bloom filter to implement the Summary Vector in our current design [Blo70]. A Bloom filter uses a vector of m bits to summarize the existence information about n fingerprints in the segment index. In `Init()`, all bits are set to 0. `Insert(a)` uses k independent hashing functions, h_1, \dots, h_k , each mapping a fingerprint a to $[0, m-1]$ and sets the bits at position $h_1(a), \dots, h_k(a)$ to 1. For any fingerprint x , `Lookup(x)` will check all bits at position $h_1(x), \dots, h_k(x)$ to see if they are all set to 1. If any of the bits is 0, then we know x is definitely not in the segment index. Otherwise, with high probability, x will be in the segment index, assuming reasonable choices of m , n , and k . Figure 3 illustrates the operations of Summary Vector.

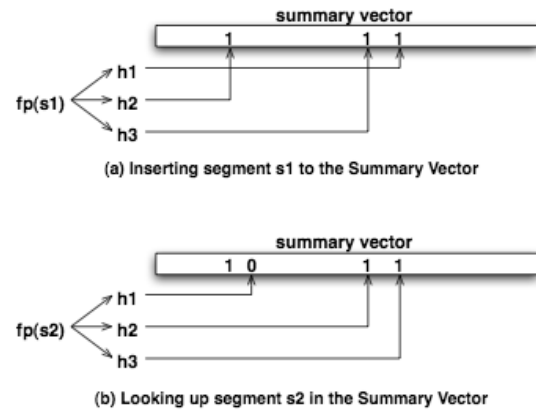


Figure 3: Summary Vector operations. The Summary Vector can identify most new segments without looking up the segment index. Initially all bits in the array are 0. On insertion, shown in (a), bits specified by several hashes, h_1 , h_2 , and h_3 of the fingerprint of the segment are set to 1. On lookup, shown in (b), the bits specified by the same hashes are checked. If any are 0, as shown in this case, the segment cannot be in the system.

As indicated in [FCAB98], the probability of false positive for an element not in the set, or the *false positive rate*, can be calculated in a straightforward fashion, given our assumption that hash functions are perfectly random. After all n elements hashed and inserted into the Bloom filter, the probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} = e^{-kn/m}.$$

The probability of false positive is then:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

Using this formula, one can derive a particular parameter to achieve a given false positive rate. For example, to achieve 2% false positive rate, the smallest size of the Summary Vector is $8 \times n$ bits ($m/n = 8$) and the number of hash functions can be 4 ($k = 4$).

To have a fairly small probability of false positive such as a fraction of a percent, we choose m such that m/n is about 8 for a target goal of n and k around 4 or 5. For example, supporting one billion base segments requires about 1 GB of memory for the Summary Vector.

At system shutdown the system writes the Summary Vector to disk. At startup, it reads in the saved copy. To handle power failures and other kinds of unclean shutdowns, the system periodically checkpoints the

Summary Vector to disk. To recover, the system loads the most recent checkpoint of the Summary Vector and then processes the containers appended to the container log since the checkpoint, adding the contained segments to the Summary Vector.

Although several variations of Bloom filters have been proposed during the past few years [BM05], we have chosen the basic Bloom Filter for simplicity and efficient implementation.

4.2 Stream-Informed Segment Layout

We use Stream-Informed Segment Layout (SISL) to create spatial locality for both segment data and segment descriptors and to enable Locality Preserved Caching as described in the next section. A stream here is just the sequence of bytes that make up a backup image stored in a Content Store object.

Our main observation is that in backup applications, segments tend to reappear in the same of very similar sequences with other segments. Consider a 1 MB file with a hundred or more segments. Every time that file is backed up, the same sequence of a hundred segments will appear. If the file is modified slightly, there will be some new segments, but the rest will appear in the same order. When new data contains a duplicate segment x , there is a high probability that other segments in its locale are duplicates of the neighbors of x . We call this property segment duplicate locality. SISL is designed to preserve this locality.

Content Store and Segment Store support a *stream* abstraction that segregates the segments created for different objects, preserves the logical ordering of segments within the Content Store object, and dedicates containers to hold segments for a single stream in their logical order. The metadata sections of these containers store segment descriptors in their logical order. Multiple streams can be written to Segment Store in parallel, but the stream abstraction prevents the segments for the different streams from being jumbled together in a container.

The design decision to make the deduplication storage system stream aware is a significant distinction from other systems such as Venti.

When an object is opened for writing, Content Store opens a corresponding stream with Segment Store which in turn assigns a container to the stream. Content Store writes ordered batches of segments for the object to the stream. Segment Store packs the new segments into the data section of the dedicated container, performs a variation of Ziv-Lempel compression on the data section, and writes segment descriptors into the metadata section of the container. When the container fills up, it appends it with Container Manager and starts a new container for

the stream. Because multiple streams can write to Segment Store in parallel, there may be multiple open containers, one for each active stream.

The end result is Stream-Informed Segment Layout or SISL, because for a data stream, new data segments are stored together in the data sections, and their segment descriptors are stored together in the metadata section.

SISL offers many benefits.

- When multiple segments of the same data stream are written to a container together, many fewer disk I/Os are needed to reconstruct the stream which helps the system achieve high read throughput.
- Descriptors and compressed data of adjacent new segments in the same stream are packed linearly in the metadata and data sections respectively in the same container. This packing captures duplicate locality for future streams resembling this stream, and enables Locality Preserved Caching to work effectively.
- The metadata section is stored separately from the data section, and is generally much smaller than the data section. For example, a container size of 4 MB, an average segment size of 8 KB, and a Ziv-Lempel compression ratio of 2, yield about 1K segments in a container, and require a metadata section size of just about 64 KB, at a segment descriptor size of 64 bytes. The small granularity on container metadata section reads allows Locality Preserved Caching in a highly efficient manner: 1K segments can be cached using a single small disk I/O. This contrasts to the old way of one on-disk index lookup per segment.

These advantages make SISL an effective mechanism for deduplicating multiple-stream fine-grained data segments. Packing containers in a stream aware fashion distinguishes our system from Venti and many other systems.

4.3 Locality Preserved Caching

We use *Locality Preserved Caching* (LPC) to accelerate the process of identifying duplicate segments.

A traditional cache does not work well for caching fingerprints, hashes, or descriptors for duplicate detection because fingerprints are essentially random. Since it is difficult to predict the index location for next segment without going through the actual index access again, the miss ratio of a traditional cache will be extremely high.

We apply LPC to take advantage of segment duplicate locality so that if a segment is a duplicate, the base segment is highly likely cached already. LPC is achieved

by combining the container abstraction with a segment cache as discussed next.

For segments that cannot be resolved by the Summary Vector and LPC, we resort to looking up the segment in the segment index. We have two goals on this retrieval:

- Making this retrieval a relatively rare occurrence.
- Whenever the retrieval is made, it benefits segment filtering of future segments in the locale.

LPC implements a segment cache to cache likely base segment descriptors for future duplicate segments. The segment cache maps a segment fingerprint to its corresponding container ID. Our main idea is to maintain the segment cache by groups of fingerprints. On a miss, LPC will fetch the entire metadata section in a container, insert all fingerprints in the metadata section into the cache, and remove all fingerprints of an old metadata section from the cache together. This method will preserve the locality of fingerprints of a container in the cache.

The operations for the segment cache are:

- `Init()`: Initialize the segment cache.
- `Insert(container)`: Iterate through all segment descriptors in container metadata section, and insert each descriptor and container ID into the segment cache.
- `Remove(container)`: Iterate through all segment descriptors in container metadata section, and remove each descriptor and container ID from the segment cache.
- `Lookup(fingerprint)`: Find the corresponding container ID for the fingerprint specified.

Descriptors of all segments in a container are added or removed from the segment cache at once. Segment caching is typically triggered by a duplicate segment that misses in the segment cache, and requires a lookup in the segment index. As a side effect of finding the corresponding container ID in the segment index, we prefetch all segment descriptors in this container to the segment cache. We call this Locality Preserved Caching. The intuition is that base segments in this container are likely to be checked against for future duplicate segments, based on segment duplicate locality. Our results on real world data have validated this intuition overwhelmingly.

We have implemented the segment cache using a hash table. When the segment cache is full, containers that are ineffective in accelerating segment filtering are leading candidates for replacement from the segment cache. A reasonable cache replacement policy is Least-Recently-Used (LRU) on cached containers.

4.4 Accelerated Segment Filtering

We have combined all three techniques above in the segment filtering phase of our implementation.

For an incoming segment for write, the algorithm does the following:

- Checks to see if it is in the segment cache. If it is in the cache, the incoming segment is a duplicate.
- If it is not in the segment cache, check the Summary Vector. If it is not in the Summary Vector, the segment is new. Write the new segment into the current container.
- If it is in the Summary Vector, lookup the segment index for its container Id. If it is in the index, the incoming segment is a duplicate; insert the metadata section of the container into the segment cache. If the segment cache is full, remove the metadata section of the least recently used container first.
- If it is not in the segment index, the segment is new. Write the new segment into the current container.

We aim to keep the segment index lookup to a minimum in segment filtering.

5 Experimental Results

We would like to answer the following questions:

- How well does the deduplication storage system work with real world datasets?
- How effective are the three techniques in terms of reducing disk I/O operations?
- What throughput can a deduplication storage system using these techniques achieve?

For the first question, we will report our results with real world data from two customer data centers. For the next two questions, we conducted experiments with several internal datasets. Our experiments use a Data Domain DD580 deduplication storage system as an NFS v3 server [PJSS*94]. This deduplication system features two-socket dual-core CPU's running at 3 Ghz, a total of 8 GB system memory, 2 gigabit NIC cards, and a 15-drive disk subsystem running software RAID6 with one spare drive. We use 1 and 4 backup client computers running NFS v3 client for sending data.

5.1 Results with Real World Data

The system described in this paper has been used at over 1,000 data centers. The following paragraphs report the deduplication results from two data centers, generated from the auto-support mechanism of the system.

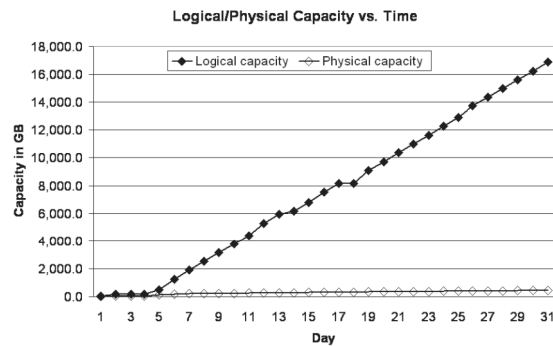


Figure 4: Logical/Physical Capacities at Data Center A

	Min	Max	Average	Standard deviation
Daily global compression	10.05	74.31	40.63	13.73
Daily local compression	1.58	1.97	1.78	0.09

Table 1: Statistics on Daily Global and Daily Local Compression Ratios at Data Center A

Data center A backs up structured database data over the course of 31 days during the initial deployment of a deduplication system. The backup policy is to do daily full backups, where each full backup produces over 600 GB at steady state. There are two exceptions:

- During the initial seeding phase (until 6th day in this example), different data or different types of data are rolled into the backup set, as backup administrators figure out how they want to use the deduplication system. A low rate of duplicate segment identification and elimination is typically associated with the seeding phase.
- There are certain days (18th day in this example) when no backup is generated.

Figure 4 shows the logical capacity (the amount of data from user or backup application perspective) and the physical capacity (the amount of data stored in disk media) of the system over time at data center A.

At the end of 31st day, the data center has backed up about 16.9 TB, and the corresponding physical capacity is less than 440 GB, reaching a total compression ratio of 38.54 to 1.

Figure 5 shows daily global compression ratio (the daily rate of data reduction due to duplicate segment elimination), daily local compression ratio (the daily rate of data reduction due to Ziv-Lempel style compression

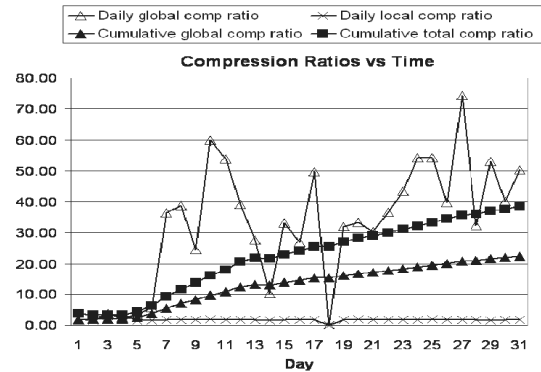


Figure 5: Compression Ratios at Data Center A

on new segments), cumulative global compression ratio (the cumulative ratio of data reduction due to duplicate segment elimination), and cumulative total compression ratio (the cumulative ratio of data reduction due to duplicate segment elimination and Ziv-Lempel style compression on new segments) over time.

At the end of 31st day, cumulative global compression ratio reaches 22.53 to 1, and cumulative total compression ratio reaches 38.54 to 1.

The daily global compression ratios change quite a bit over time, whereas the daily local compression ratios are quite stable. Table 1 summarizes the minimum, maximum, average, and standard deviation of both daily global and daily local compression ratios, excluding seeding (the first 6 days and no backup (18th day).

Data center B backs up a mixture of structured database and unstructured file system data over the course of 48 days during the initial deployment of a deduplication system using both full and incremental backups. Similar to that in data center A, seeding lasts until the 6th day, and there are a few days without backups (8th, 12-14th, 35th days). Outside these days, the maximum daily logical backup size is about 2.1 TB, and the smallest size is about 50 GB.

Figure 6 shows the logical capacity and the physical capacity of the system over time at data center B.

At the end of 48th day, the logical capacity reaches about 41.4 TB, and the corresponding physical capacity is about 3.0 TB. The total compression ratio is 13.71 to 1.

Figure 7 shows daily global compression ratio, daily local compression ratio, cumulative global compression ratio, and cumulative total compression ratio over time.

At the end of 48th day, cumulative global compression reaches 6.85, while cumulative total compression reaches 13.71.

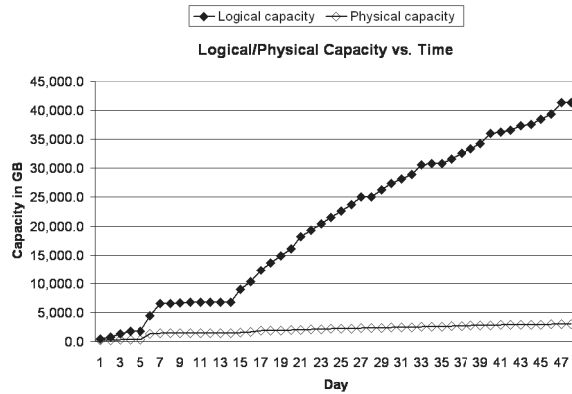


Figure 6: Logical/Physical Capacities at Data Center B.

	Min	Max	Average	Standard deviation
Daily global compression	5.09	45.16	13.92	9.08
Daily local compression	1.40	4.13	2.33	0.57

Table 2: Statistics on Daily Global and Daily Local Compression Ratios at Data Center B

	Exchange data	Engineering data
Logical capacity (TB)	2.76	2.54
Physical capacity after deduplicating segments (TB)	0.49	0.50
Global compression	5.69	5.04
Physical capacity after local compression (TB)	0.22	0.261
Local compression	2.17	1.93
Total compression	12.36	9.75

Table 3: Capacities and Compression Ratios on Exchange and Engineering Datasets

Table 2 summarizes the minimum, maximum, average, and standard deviation of both daily global and daily local compression ratios, excluding seeding and days without backup.

The two sets of results show that the deduplication storage system works well with the real world datasets. As expected, both cumulative global and cumulative total compression ratios increase as the system holds more backup data.

During seeding, duplicate segment elimination tends to be ineffective, because most segments are new. After seeding, despite the large variation in the actual number, duplicate segment elimination becomes extremely

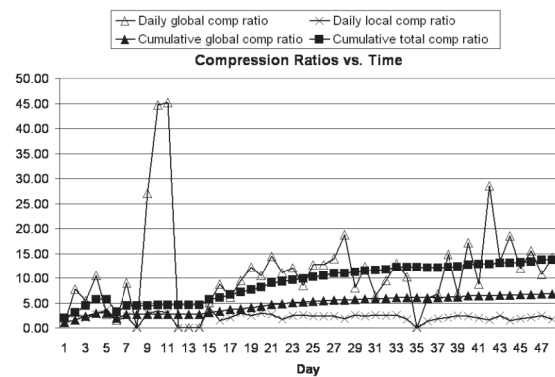


Figure 7: Compression Ratios at Data Center B.

effective. Independent of seeding, Ziv-Lempel style compression is relatively stable, giving a reduction of about 2 over time. The real world observations on the applicability of duplicate segment elimination during seeding and after seeding are particularly relevant in evaluating our techniques to reduce disk accesses below.

5.2 I/O Savings with Summary Vector and Locality Preserved Caching

To determine the effectiveness of the Summary Vector and Locality Preserved Caching, we examine the savings for disk reads to find duplicate segments using a Summary Vector and Locality Preserved Caching.

We use two internal datasets for our experiment. One is a daily full backup of a company-wide Exchange information store over a 135-day period. The other is the weekly full and daily incremental backup of an Engineering department over a 100-day period. Table 3 summarizes key attributes of these two datasets.

These internal datasets are generated from production usage (albeit internal). We also observe that various compression ratios produced by the internal datasets are relatively similar to those of real world examples examined in section 5.1. We believe these internal datasets are reasonable proxies of real world deployments.

Each of the backup datasets is sent to the deduplicating storage system with a single backup stream. With respect to the deduplication storage system, we measure the number of disk reads for segment index lookups and locality prefetches needed to find duplicates during write for four cases:

- (1) with neither Summary Vector nor Locality Preserved Caching;
- (2) with Summary Vector only;
- (3) with Locality Preserved Caching only; and

- (4) with both Summary Vector and Locality Preserved Caching.

The results are shown in Table 4.

Clearly, the Summary Vector and Locality Preserved Caching combined have produced an astounding reduction in disk reads. Summary Vector alone reduces about 16.5% and 18.6% of the index lookup disk I/Os for exchange and engineering data respectively. The Locality Preserved Caching alone reduces about 82.4% and 81% of the index lookup disk I/Os for exchange and engineering data respectively. Together they are able to reduce the index lookup disk I/Os by 98.94% and 99.6% respectively.

In general, the Summary Vector is very effective for new data, and Locality Preserved Caching is highly effective for little or moderately changed data. For backup data, the first full backup (seeding equivalent) does not have as many duplicate data segments as subsequent full backups. As a result, the Summary Vector is effective to avoid disk I/Os for the index lookups during the first full backup, whereas Locality Preserved Caching is highly beneficial for subsequent full backups. This result also suggests that these two datasets exhibit good duplicate locality.

5.3 Throughput

To determine the throughput of the deduplication storage system, we used a synthetic dataset driven by client computers. The synthetic dataset was developed to model backup data from multiple backup cycles from multiple backup streams, where each backup stream can be generated on the same or a different client computer.

The dataset is made up of synthetic data generated on the fly from one or more backup streams. Each backup stream is made up of an ordered series of synthetic data

versions where each successive version (“generation”) is a somewhat modified copy of the preceding generation in the series. The generation-to-generation modifications include: data reordering, deletion of existing data, and addition of new data. Single-client backup over time is simulated when synthetic data generations from a backup stream are written to the deduplication storage system in generation order, where significant amounts of data are unchanged day-to-day or week-to-week, but where small changes continually accumulate. Multi-client backup over time is simulated when synthetic data generations from multiple streams are written to the deduplication system in parallel, each stream in the generation order.

There are two main advantages of using the synthetic dataset. The first is that various compression ratios can be built into the synthetic model, and usages approximating various real world deployments can be tested easily in house.

The second is that one can use relatively inexpensive client computers to generate an arbitrarily large amount of synthetic data in memory without disk I/Os and write in one stream to the deduplication system at more than 100 MB/s. Multiple cheap client computers can combine in multiple streams to saturate the intake of the deduplication system in a switched network environment. We find it both much more costly and technically challenging using traditional backup software, high-end client computers attached to primary storage arrays as backup clients, and high-end servers as media/backup servers to accomplish the same feat.

In our experiments, we choose an average generation (daily equivalent) global compression ratio of 30, and an average generation (daily equivalent) local compression ratio of 2 to 1 for each backup stream. These compression numbers seem possible given the real world examples in section 5.1. We measure throughput for one

	Exchange data		Engineering data	
	# disk I/Os	% of total	# disk I/Os	% of total
no Summary Vector and no Locality Preserved Caching	328,613,503	100.00%	318,236,712	100.00%
Summary Vector only	274,364,788	83.49%	259,135,171	81.43%
Locality Preserved Caching only	57,725,844	17.57%	60,358,875	18.97%
Summary Vector and Locality Preserved Caching	3,477,129	1.06%	1,257,316	0.40%

Table 4: Index and locality reads. This table shows the number disk reads to perform index lookups or fetches from the container metadata for the four combinations: with and without the Summary Vector and with and without Locality Preserved Caching. Without either the Summary Vector or Locality Preserved Caching, there is an index read for every segment. The Summary Vector avoids these reads for most new segments. Locality Preserved Caching avoids index lookups for duplicate segments at the cost an extra read to fetch a group of segment fingerprints from the container metadata for every cache miss for which the segment is found in the index.

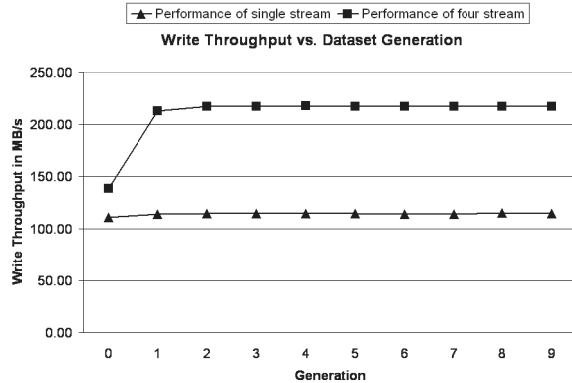


Figure 8: Write Throughput of Single Backup Client and 4 Backup Clients.

backup stream using one client computer and 4 backup streams using two client computers for write and read for 10 generations of the backup datasets. The results are shown in Figures 8 and 9.

The deduplication system delivers high write throughput results for both cases. In the single stream case, the system achieves write throughput of 110 MB/sec for generation 0 and over 113 MB/sec for generation 1 through 9. In the 4 stream case, the system achieves write throughput of 139 MB/sec for generation 0 and a sustained 217 MB/sec for generation 1 through 9.

Write throughput for generation 0 is lower because all segments are new and require Ziv-Lempel style compression by the CPU of the deduplication system.

The system delivers high read throughput results for the single stream case. Throughout all generations, the system achieves over 100 MB/sec read throughput.

For the 4 stream case, the read throughput is 211 MB/sec for generation 0, 192 MB/sec for generation 1, 165 MB/sec for generation 2, and stay at around 140 MB/sec for future generations. The main reason for the decrease of read throughput in the later generations is that future generations have more duplicate data segments than the first few. However, the read throughput stays at about 140 MB/sec for later generations because of Stream-Informed Segment Layout and Locality Preserved Caching.

Note that write throughput has historically been valued more than read throughput for the backup use case since backup has to complete within a specified backup window time period and it is much more frequent event than restore. Read throughput is still very important, especially in the case of whole system restores.

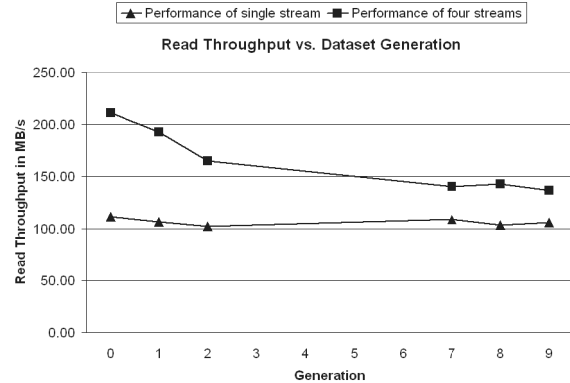


Figure 9: Read Throughput of Single Backup Client and 4 Backup Clients

5.4 Discussion

The techniques presented in this paper are general methods to improve throughput performance of deduplication storage systems. Although our system divides a data stream into content-based segments, these methods can also apply to system using fixed aligned segments such as Venti.

As a side note, we have compared the compression ratios of a system segmenting data streams by contents (about 8Kbytes on average) with another system using fixed aligned 8Kbytes segments on the engineering and exchange backup datasets. We found that the fixed alignment approach gets basically no global compression (global compression: 1.01) for the engineering data, whereas the system with content-based segmentation gets a lot of global compression (6.39:1). The main reason of the difference is that the backup software creates the backup dataset without realigning data at file boundaries. For the exchange backup dataset where the backup software aligns data at individual mailboxes, the global compression difference is less (6.61:1 vs. 10.28:1), though there is a significant gap.

Fragmentation will become more severe for long term retention, and can reduce the effectiveness of Locality Preserved Caching. We have investigated mechanisms to reduce fragmentation and sustain high write and read throughput. But, these mechanisms are beyond the scope of this paper.

6 Related Work

Much work on deduplication focused on basic methods and compression ratios, not on high throughput.

Early deduplication storage systems use file-level hashing to detect duplicate files and reclaim their storage space [ABCC*02, TSKK*03, KDLT04]. Since such

systems also use file hashes to address files. Some call such systems content addressed storage or CAS. Since their deduplication is at file level, such systems can achieve only limited global compression.

Venti removes duplicate fixed-size data blocks by comparing their secure hashes [QD02]. It uses a large on-disk index with a straightforward index cache to lookup fingerprints. Since fingerprints have no locality, their index cache is not effective. When using 8 disks to lookup fingerprints in parallel, its throughput is still limited to less than 7 MB/sec. Venti used a container abstraction to layout data on disks, but was stream agnostic, and did not apply Stream-Informed Segment Layout.

To tolerate shifted contents, modern deduplication systems remove redundancies at variable-size data blocks divided based on their contents. Manber described a method to determine anchor points of a large file when certain bits of rolling fingerprints are zeros [Man93] and showed that Rabin fingerprints [Rab81, Bro93] can be computed efficiently. Brin et al. [BDH94] described several ways to divide a file into content-based data segments and use such segments to detect duplicates in digital documents. Removing duplications at content-based data segment level has been applied to network protocols and applications [SW00, SCPC*02, RLB03, MCK04] and has reduced network traffic for distributed file systems [MCM01, JDT05]. Kulkarni et al. evaluated the compression efficiency between an identity-based (fingerprint comparison of variable-length segments) approach and a delta-compression approach [KDLT04]. These studies have not addressed deduplication throughput issues.

The idea of using Bloom filter [Blo70] to implement the Summary Vector is inspired by the summary data structure for the proxy cache in [FCAB98]. Their work also provided analysis for false positive rate. In addition, Broder and Mitzenmacher wrote an excellent survey on network applications of Bloom filters [AM02]. TAPER system used a Bloom filter to detect duplicates instead of detecting if a segment is new [JDT05]. It did not investigate throughput issues.

7 Conclusions

This paper presents a set of techniques to substantially reduce disk I/Os in high-throughput deduplication storage systems.

Our experiments show that the combination of these techniques can achieve over 210 MB/sec for 4 multiple write data streams and over 140 MB/sec for 4 read data streams on storage server with two dual-core processors and one shelf of 15 drives.

We have shown that Summary Vector can reduce disk index lookups by about 17% and Locality Preserved Caching can reduce disk index lookups by over 80%, but the combined caching techniques can reduce disk index lookups by about 99%.

Stream-Informed Segment Layout is an effective abstraction to preserve spatial locality and enable Locality Preserved Caching.

These techniques are general methods to improve throughput performance of deduplication storage systems. Our techniques for minimizing disk I/Os to achieve good deduplication performance match well against the industry trend of building many-core processors. With quad-core CPU's already available, and eight-core CPU's just around the corner, it will be a relatively short time before a large-scale deduplication storage system shows up with 400 ~ 800 MB/sec throughput with a modest amount of physical memory.

8 References

- [ABCC*02] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, December 2002.
- [BM05] Andrie Z. Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 2005.
- [BDH94] S. Brin, J. Davis, H. Garcia-Molina. Copy Detection Mechanisms for Digital Documents (weblink). 1994, also Iso in *Proceedings of ACM SIGMOD*, 1995.
- [Blo70] Burton H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13 (7). 422-426.
- [JDT05] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proceedings of USENIX File And Storage Systems (FAST)*, 2005.
- [Dat05] Data Domain, Data Domain Appliance Series: High-Speed Inline Deduplication Storage, 2005, <http://www.datadomain.com/products/appliances.htm>
- [FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrie Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. in *Proceedings of ACM SIGCOMM'98*, (Vancouver, Canada, 1998).
- [KDLT04] P. Kulkarni, F. Douglass, J. D. LaVoie, J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of USENIX Annual Technical Conference*, pages 59-72, 2004.

- [Man93] Udi Manber. Finding Similar Files in A Large File System. Technical Report TR 93-33, Department of Computer Science, University of Arizona, October 1993, also in *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 17–21. 1994.
- [MCK04] J. C. Mogul, Y.-M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of Network Systems Design and Implementation*, 2004.
- [MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-bandwidth Network File System. In *Proceedings of the ACM 18th Symposium on Operating Systems Principles*. Banff, Canada. October, 2001.
- [NIST95] National Institute of Standards and Technology, FIPS 180-1. Secure Hash Standard. US Department of Commerce, April 1995.
- [PJSS*94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, NFS Version 3 Design and Implementation, In *Proceedings of the USENIX Summer 1994 Technical Conference*. 1994.
- [QD02] S. Quinlan and S. Dorward, Venti: A New Approach to Archival Storage. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)*, January 2002.
- [RLB03] S. C. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *WWW*, pages 619–628, 2003.
- [SCPC*02] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of USENIX Operating Systems Design and Implementation*, 2002.
- [SW00] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, pages 87–95, Aug. 2000.
- [TKSK*03] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.
- [YPL05] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '05)*, April 2005.
- [ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory*, vol. IT-23, pp. 337–343, May 1977.

Towards Tamper-evident Storage on Patterned Media

Pieter H. Hartel Leon Abelmann Mohammed G. Khatib
*Fac. of Electrical Engineering, Mathematics and Computer Science,
Univ. of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands,
email {p.h.hartel, l.abelmann, m.g.khatib}@utwente.nl*

Abstract

We propose a tamper-evident storage system based on probe storage with a patterned magnetic medium. This medium supports normal read/write operations by out-of-plane magnetisation of individual magnetic dots. We report on measurements showing that in principle the medium also supports a separate class of write-once operation that destroys the out-of-plane magnetisation property of the dots irreversibly by precise local heating. We discuss the main issues of designing a tamper-evident storage device and file system using the properties of the medium.

1 Introduction

Tampering with data is a problem because it affects many businesses and organisations [11]. The culprits include everyone from the casual hacker who tries to obliterate the trace of his actions, to the CEO and his accountant who destroy vital evidence [10]; and the stakes are high [20]. As a result, laws such as the US Sarbanes-Oxley Act of 2002 (SOX) [37] and the EU data retention directive [49] have been introduced to create a legal framework in which to deal with tampering.

To deal with tampering on a technical level a large variety of disk, tape, and optical Write Once Read Many (WORM) technologies have been developed that are designed [15] to resist or at least to help detect tampering. Unfortunately, tamper resistance is a hard problem [3] that none of the existing storage technologies actually provide. For example, there is nothing to stop a dedicated attacker from tampering with a Read-only Memory (ROM) [4]. On the other hand it is difficult to cover the traces of tampering with a ROM. Mass storage such as disk, tape and optical disk is probably just as easy to tamper with as ROM, so we believe that we should put our efforts in improving the tamper evidence of mass storage.

It is always difficult to find the right balance between

security and usability, and tamper evidence of mass storage is no exception to this rule. To guide the discussion we analyse how mass storage is used. Probably most applications use a data base, which requires efficient random reads and writes. Tampering with data is easy, as any record can simply be rewritten. Therefore most data bases support a snapshot operation that freezes the contents of the data base, for instance for auditing purposes, recovery, etc. If the snapshot is written to a disk, the attacker will find it as easy to tamper with the snapshot as it is easy to tamper with the live database. If on the other hand the snapshot is written to an optical WORM device, tampering would be more difficult to hide. Unfortunately the WORM device also has an impact on the way the snapshot can be read, as the performance characteristics of hard disk and optical disk are different. An ideal solution would combine the performance of the hard disk with the tamper evidence of the optical disk. In other words we require not a WORM device but a Selectively Eventually Read-only (SERO) device, i.e., a device that begins life as a Write Many Read Many (WORM) device, selected parts of which are subjected to Write Once (WO) operations, and which ends life as a Read-only (RO) device.

The construction of a SERO device is still a long way away, but in this paper we would like to lay the physical foundation for the medium used in such a device, also giving the design considerations for a device and a file system that would support SERO storage. Our proposal combines an idea of Molnar et al. [31] for *tamper-evident storage* with our own work on *patterned media*. We introduce each of these two elements below.

Molnar et al. [31] describe how standard Programmable Read-Only Memory (PROM) can be used to build *tamper-evident storage*. The basic idea is to store each bit of information in a cell occupying two bits using Manchester encoding: the logical bit 1 is encoded as the cell 10 and the logical bit 0 is encoded as the cell 01. The value 11 indicates a cell that has not yet been used

(all cells in a PROM are initialized to 11). The value 00 indicates a cell that has been tampered with for the following reason. The physical properties of a PROM make it *impossible* to change a 0 back into a 1 (except by exposing the entire memory module to ultra violet light, which would reset all cells to 11). Therefore, the only way to tamper with information (which is encoded as 01 or 10) is to clear a bit. This immediately results in an invalid cell 00, which provides the evidence of tampering.

A *patterned medium* [52] consists of a regular arrangement of magnetic dots separated by sub-micron distances that can be magnetised in two directions along a fixed magnetic axis. A magnetic dot can be read and written magnetically any number of times. However, by precise local heating of a dot, the orientation of the magnetic axis of the dot can be changed irreversibly. The idea is to use this feature to create a storage device that begins life as a WORM device, reading and writing dots magnetically. After heating, a dot can no longer be read or written magnetically, but the fact that a dot has been heated can be detected. From then on the heated parts of the medium operate as a tamper-evident RO device, while the rest of the medium continues to operate as a WORM device. The ability to heat parts of the medium incrementally provides flexibility that cannot be matched by current WORM technology. (The operation we call “heating” is usually called “freezing” in the literature, but given the physical realisation of the operation we decided to stay with the term heating.)

In the rest of the paper, we discuss the issues that must be addressed for the combination of tamper-evident storage and patterned media to result in a SERO device with the following properties. Firstly, like a hard disk, the device is expected to offer random WORM access to a large number of blocks with a total capacity of the order of 1 Terabit [39]. Secondly, the device is expected to be capable of a WO operation of a block by heating the magnetic dots of the block. After the WO operation the block is RO. We will also refer to the WO operation of a block as *heating a block*. Finally, heating a block is expected to be relatively slow. Therefore, the device is expected to be able to heat a contiguous sequence of blocks, henceforth referred to as *heating a line*, by (a) calculating a secure hash of the line, and (b) applying a WO operation on the first block of the line to record the hash.

Our proposal does not use cryptographic keys. We provide only data integrity (using secure hashing and hardware support) but no confidentiality or authenticity. Our proposal is thus complementary to the vast amount of work on using symmetric and public key cryptography to provide storage with confidentiality and authenticity, and our work could be combined with many of the existing approaches.

Contribution The contributions of the paper are (1) to evaluate the feasibility of heating dots, and (2) to discuss whether a tamper-evident probe storage device and file system on a patterned medium are feasible. The paper touches upon all the relevant aspects from regulatory issues studied by lawyers down to material science studied by physicists.

Related work is discussed in the next section. Then we speculate on the feasibility of a device (Section 3), and a file system (Section 4) for SERO storage. A security analysis of the hypothetical SERO file system and device is presented in Section 5. We describe concrete examples of an actuator (Section 6), and a medium (Section 7) that could be used to build a SERO device. Section 8 discusses open issues. The last section concludes and suggests further work.

2 Related work

We discuss related work in a top down fashion, starting with the regulatory issues all the way down to material science.

Regulatory issues The world of data storage goes through a period of turmoil. Taylor [49] describes how the EU data retention directive will increase the difficulty of companies and organisations to comply with the already burdensome laws and regulations. Hasan et al. [17] describe the struggles and demise of Storage Service Providers, largely due to regulations such as SOX, while the business case for outsourcing data storage is as strong as ever due to the rising Total Cost of Ownership (TCO). Hasan and Yurcik [16] discuss the effects of disclosure legislation on companies, which stipulates that storage security breaches must be reported, in some cases even in public, on TV and in newspapers. The effect on the reputation of businesses affected can be devastating. Tamper-evident storage is needed to help address the problems.

Tamper evidence There are three basic approaches to providing tamper evidence. The first and most commonly practiced approach relies on hardware support; for example using Write Once Read Many (WORM) technology [5]. The main disadvantage is that WORM technology tends to be inflexible; data can only be written once, while most applications (chiefly data bases) write and rewrite data often until the moment has arrived to take a snapshot for auditing and compliance purposes. The second approach to providing tamper evidence relies on a trusted third party (TTP) to provide notary services [6], secure time stamps [26], etc. It is not always

practical to rely on a TTP; for example in mobile applications the TTP may not always be reachable. The third approach either (a) distributes the data over many servers, only some of which are assumed to be malicious or faulty [2], or (b) uses many clients to control the server. A good example of the latter is SUNDR [24], in which each client keeps a record of his last transaction with the SUNDR server. This allows the client to check whether his previous transaction has somehow been “forgotten” by the server. This works fine as long as all clients regularly check their last transaction, but the mechanism is ineffective if this is not the case; for example if most clients make only one transaction. SUNDR is geared towards detecting tampering during data sharing; in our work we do rely on others to detect tampering but rely on hardware support.

All the approaches mentioned rely on something external to system that is intended to deliver the secure store (i.e., separate hardware, separate servers and/or separate clients). Our approach relies on hardware support, while improving the flexibility beyond what a typical WORM system can offer.

WORM technologies Write-protect rings have been used for 50 years on magnetic tapes to prevent accidental overwriting of valuable data. There are many variations on this idea to protect disks, tapes and optical media.

Physical WORM technologies using optical media and tape are widely used. Optical WORM technologies offer a high level of integrity but the cost of ownership is higher than that of disk-based technologies. For example CDROMs are cumbersome to manage (because they have a small capacity, which leads to large collections of CDs), and professional optical storage systems are expensive (because they often contain mechanical robots). Tape based technologies are generally inexpensive but offer integrity at the medium level only. For example a tape cartridge in the Linear Tape-Open 3 (LTO-3) industry standard has a small semiconductor memory in which a read-only flag can be set [21], such that a compliant tape drive will refuse to write on such a cartridge. The tape itself can still be written using a tape drive that has been tampered with, or after tampering with the cartridge.

Software based WORM technologies are based on the idea that the disk driver or the firmware of the disk can be modified to block future writes to selected areas of the disk. The integrity offered by this approach is relatively weak, as software modifications can generally be undone. There are many Virtual Tape Library products in the market that depend on software based WORM technologies [55].

An IBM patent [56] proposes to connect the write signal of a disk head via a blowable fuse such that once the

fuse is blown, an entire disk platter becomes immutable. This offers a high level of relatively coarse grained integrity. As in the case of the LTO-3 tape standard, the platter is still writeable but it would be more difficult to repair the fuse in the head than it is to tamper with an LTO-3 tape drive.

Probe storage During the last ten years, several recording systems based on probe microscopy technology have been proposed. Leading research by IBM [39] is followed by other companies such as HP [34], Samsung [30], Seagate [23], LG [22] and a number of universities such as Carnegie Mellon, DSI Singapore, Exeter, Tohoku, Twente, and Yonsei. Probe storage is also being combined with disk storage [18].

Materials aspects To understand the details of the modification of magnetic materials, background information is given in section 6. For the following it is sufficient to understand that the individual elements in the patterned medium, the dots, have an easy direction of magnetisation perpendicular to the film surface, rather than in-plane. This is achieved by using a stack of ultra-thin films (tens of layers, each thinner than 1 nm) of interleaved magnetic and non-magnetic material. The many interfaces between the magnetic and non-magnetic films force the magnetisation perpendicular to those interfaces, and therefore to the film.

The modification of the magnetic properties of a multilayered patterned medium is relatively easy. The first experiments were performed with Ga ions from a Focused Ion Beam [50], using modest irradiation doses. The magnetic properties of the material are modified by displacement of the interface atoms, and inclusion of Ga. By using lighter ions, such as He^+ , the incorporation of ions can be avoided, and only interface mixing results [41]. As a result, the easy axis of magnetisation rotates from perpendicular to in-plane. By using shadow masks to shield from the impingement of ions, these irradiation techniques can be used to pattern multilayered films into areas with perpendicular and in-plane magnetisation. These types of patterned medium have the advantage that the surface remains flat.

In this work we suggest to use temperature-assisted interface mixing to destroy magnetic dots selectively. On the effect of heat treatment of multilayer materials, much less is known, primarily because it is considered a detrimental effect that cannot be used for patterning. Encouraging experiments show however that at relatively low temperatures of about 300 °C, interface mixing occurs between Co (magnetic) and Pt (non-magnetic) [46]. Heat treatment can however also have beneficial effects on the interfaces. In Co/Cu systems for instance, the interfaces are found to enhance at temperatures of 300 °C [7]. Most

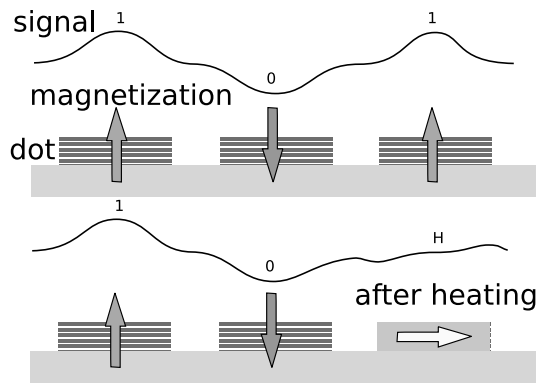


Figure 1: Above: dots are magnetised upwards or downwards; Below: destroyed dots have a perpendicular or in-plane easy axis.

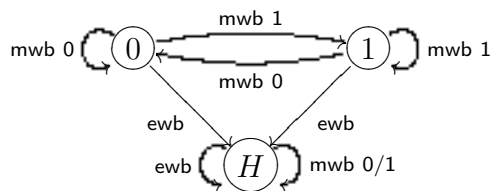


Figure 2: The state transitions of one bit. *H* indicates a heated bit, and 0/1 indicates a bit that has not been heated.

likely this has to do with the solubility of both materials. Therefore the material combination has to be chosen with care. Even so, it is possible to damage the films at higher temperatures. In the same Co/Cu system, at 700 °C grains start to grow and the Co layers start to coalesce, thus destroying the interface completely. From these experiments we can conclude therefore that thermal destruction of the magnetic properties by interface mixing is in principle possible, provided that the right material combination is chosen.

This concludes the survey of related work on all relevant aspects of tamper-evident data storage.

3 Device

We argue that tamper evidence storage requirements can be served flexibly by six high level sector operations, which are built out of four low level bit operations. We describe the bit operations first, followed by the sector operations.

Magnetic bit operations We require magnetic read and write operations for bits and a second set of electrical read and write operations for bits. We discuss the

magnetic read and write operations first. In the normal mode of operation we have a medium with a regular matrix of magnetic single domain dots with a preferred axis of magnetisation perpendicular to the medium. This is illustrated in the top half of Figure 1, which shows the substrate and three layered dots. The first and last are magnetised in the upwards direction, the middle in the downwards direction.

The magnetic write bit operation *mwb* sets the direction of the magnetisation (up is 1, down is 0) and the corresponding magnetic read bit operation *mr* senses the direction of the magnetisation. The signal measured by the read heads is shown schematically, indicating a positive peak for the first and the last dot, and a negative peak for the middle dot. The top half of Figure 2 illustrates the transitions from 0 to 1 and vice versa, as effectuated by the magnetic write operations on the state of an individual bit.

Electrical bit operations The second set of read and write operations on bits has an electrical basis. The electrical write bit operation *ewb* heats an individual dot by means of an electric current flowing from the probe tip via the dot to the medium. This heating causes the multi-layer structure of a dot to be destroyed and as a result the easy axis of magnetisation rotates into the medium. The data stored by magnetic write operations is lost. Now, we have a third way of representing data, indicated by the letter *H* (for heated), and effectuated by the *ewb* operation. (We will also indicate un-heated bits by the letter *U*). The bottom half of Figure 1 shows that the layered structure of the last dot is permanently destroyed. The peak in the magnetic read signal for the last bit has disappeared. The electrical write bit operation *ewb* is an irreversible process, which can only change a 0/1 bit into a *H*, as shown by the one-way transitions from the states 0 and 1 in the top half of Figure 2 to the state *H*. (See Section 7 for detail on the physics.)

Strictly speaking there is no electrical read bit operation *erb*; instead *erb* is built out of magnetic read and write operations. The operation *erb* detects the presence or absence of an out-of-plane dot by performing an atomic sequence of *mr* and *mwb* operations as follows:

1. *mr* to read the original bit;
2. *mwb* to write the inverse of the original bit;
3. *mr* of the inverse to verify that the inverse can indeed be read back;
4. *mwb* to write the original again;
5. *mr* to verify that the original can indeed be read back.

Block	Bit number					Purpose
	0	1	...	4094	4095	
0	HU/UH		...	HU/UH		hash+meta.
1	0/1	0/1	...	0/1	0/1	512B data
2	0/1	0/1	...	0/1	0/1	512B data
$2^N - 1$			⋮			
	0/1	0/1	...	0/1	0/1	512B data

Figure 3: Sample medium layout of a heated line of 512 bytes=4096 bits each. 0/1 represents a magnetically written bit, HU represents an electrically written, Manchester encoded logical 0 and UH represents a logical 1.

If any of the two verification steps fail we assume that the dot has lost its out-of-plane property and let the electrical read operation erb return H , else erb returns U (the two inversions ensure that the original magnetic data is restored for dots that have not been heated).

The erb operation is at least 5 times slower than mr_b , and ew_b is also slower than mw_b because of the local heating process. Therefore, as stated before, the idea is to use the erb and ew_b operations sparingly.

As illustrated in Figure 2 (bottom right), applying a single mr_b operation to an electrically written bit would yield a more or less random result. To avoid this, the device must follow the proper protocol which means that magnetically written data must only be read magnetically and that electrically written data must only be read electrically. A simple way to achieve this is by reserving specific physical areas for electrical data while using other areas for magnetic data. As we shall see below, this rigid segregation of electrical and magnetic data puts severe constraints on the design of the device and the file system. An alternative would be to read the in-plane magnetic signal directly, however, this requires carefully constructed elliptic dots to ensure that the direction of the in-plane magnetization is known (See Section 7).

Sector operations Following Pozidis et al. [39] we assume that a sector has a standard size of 512 bytes and about 15% sector overhead for the sector header, error correction, and cyclic redundancy check. This allows us to build a magnetic read sector operation mrs and a magnetic write sector operation mws using the magnetic read and write operations for bits described above, and taking error correction appropriate to the medium, the tips, etc. into account. Henceforth we will talk about a block as the smallest unit of storage, and for simplicity we assume that a block is a single sector. Similarly we can build an electrical read sector operation ers and an electrical write sector operation ews using the electrical read and write operations for bits described above.

Heat a line Assume that at a certain moment some existing data must be heated, after which the data cannot be destroyed without leaving a trace. Our heat operation works on a *line*, which is a sequence of 2^N contiguous blocks aligned on a 2^N boundary. When given a line, the heat operation performs the following atomic sequence of steps:

1. Read blocks $1 \dots 2^N - 1$ representing the line to be protected using $2^N - 1$ calls to mrs ;
2. Calculate a secure hash (e.g., SHA-256) of the blocks and their addresses just read;
3. Write the 512-bit Manchester encoding of the 256-bit hash in block 0 using the electrical write operation ews , this leaves $4096 - 512 = 3584$ bits of space for meta data, signatures, etc.;
4. Check that the hash can be read back using ers , or else fail.

All lines can be heated individually, thus providing significant flexibility over WORM-based approaches. Blocks $1 \dots 2^N - 1$ of a heated line can still be read magnetically, hence efficiently, and as often as needed. Figure 3 illustrates the result of the heat operation. The last $2^N - 1$ blocks are written magnetically, shown as zeros and ones. Block 0 is written electrically in a Manchester encoding, where each logical bit of the hash occupies two physical bits on the medium. The Manchester encoding ensures that a heated bit (i.e., an H) has at most one heated neighbour. Since each electrical write may be expected to have a detrimental effect on the neighbouring bits, spreading out heated bits is good for reliability.

The heat operation, when applied to a line that has already been heated either has no effect and is therefore harmless (if the data in block 0 is invariant) or it will turn Manchester encoded bits into HH , thus providing evidence of tampering.

Verify a heated line The verify operation computes the hash of a line and compares the computed hash to the electrically written hash. A mismatch represents evidence of tampering.

Addressing Modern disks offer a uniform method of accessing blocks by logical block address, rather than by physical block addresses (which may vary wildly between devices), and automatic bad block handling by the device offers the file system the abstraction of a reliable device. However, to be tamper-evident we must know exactly where to look for evidence of tampering. This means that a SERO device and the SERO file system should use physical block addresses (PBA) rather than

logical block addresses (LBA) so that we know exactly at which PBA to look for heated hashes. Bad block handling is a challenge, because a heated block should not be misinterpreted as a bad block. If the disk exposes its physical layout to the file system, the file system should be able to recognize when data is in the right place.

4 File system

Having described a SERO device that can make a line RO by heating the line, we discuss the main questions that the designer of a file system must address to serve such a device. The main question that we wish to pose is *what properties a high performance, tamper-evident file system should have so that it can serve a SERO device*. We will explore the performance issues first, followed by the tamper evidence issues.

4.1 SERO file system performance

Standard hard disk WORM storage offers high data rates and low access times, whereas WORM storage typically has higher access times (especially when tape or disk robots are involved), and lower data rates than WORM. A SERO mass storage device combines the two classes of use in one device, which poses a challenge to the device and the file system not to degrade the performance of WORM operations due to the presence of RO lines. The two types of storage are normally served by different file systems, whereas a SERO device could probably be served better by a single file system.

As a SERO device ages, slowly but surely parts of the storage become RO, such that the WORM area not only shrinks but it might also become fragmented. Considering that it does not make sense to move a RO line (because this would not leave behind usable space), the file system has an important task in avoiding fragmentation of heated lines.

Interestingly, part of the answer to the question we posed at the beginning of this section is provided by Rosenblum and Ousterhout, who observe that when the read cache is large enough, disk I/O is dominated by writes. Therefore, the disk has the best chance of keeping up with the CPU if writes are clustered [42]. Many file systems have since been proposed that cluster writes. From a write performance point of view it makes no difference whether the blocks in a cluster are related, for instance when the blocks are part of the same file or when the blocks are unrelated. However, from the SERO point of view it does make a difference whether blocks are related, because it does not make sense to heat a line of unrelated blocks. In the end, it depends on the application whether or not clusters of related blocks are likely

to occur. For instance, taking a data base snapshot would probably result in a cluster of related blocks.

So why does clustering help our SERO device? Clustering makes it possible to take a contiguous sequence of related blocks, to hash the data stored in those blocks, and to use the WO operation to store the hash of the sequence. The advantages of clustering are twofold. Firstly, the larger the cluster, the lower the overhead of the hash can be. Secondly, the WO operation is expected to be considerably slower than the WM operation, and clustering allows the WO operation to be used sparingly.

We will now have a closer look at the original log-structured file system [42]. LFS treats the space on the disk as a collection of contiguous segments, each of which consists of a contiguous sequence of blocks. This collection of segments is called the log. New data is written sequentially to the log and the log is filled incrementally.

An LFS has to manage data blocks and free blocks on the storage device, while keeping the performance of the disk as high as possible. To achieve this performance goal, it (1) accumulates small writes and commits them to the disk in a single operation, and (2) gathers related but scattered blocks, removing dirty blocks by running the garbage collector.

The presence of heated lines complicates the tasks of the LFS. This is because once a line has been heated it cannot be copied by the garbage collector, since a heated line leaves no reusable space behind. Copying a heated line just decreases the free space that can be potentially used for new data. Therefore, like clustering of related blocks, heated lines should also be clustered.

Based on the behaviour of the application, it should be possible to predict which lines will be heated at the same time. Therefore, during garbage collection, the file system may cluster lines into segments, that are likely to be heated at the same time. As a result of such a clustering policy, the file system creates a bimodal distribution of heated segments; that is we have only mostly heated segments and mostly unheated segments. As a result, (1) the performance of reading/writing blocks should not be affected much, since heated lines and WORM live data blocks are kept separate, (2) space decreases only if new data is written and not when lines are heated, since lines are heated in the right place, avoiding the need to copy them, and (3) the garbage collector skips over heated segments, avoiding reading and writing them repeatedly, thus saving on disk bandwidth. Summarizing, the bimodality should help to keep the performance high in the presence of heated lines.

Other file systems do not use a log, but pack data into clusters. For example, the Berkeley Fast File Systems (FFS) [44] uses clusters to pack small files with their metadata, or to pack related blocks of large files into the

same cluster. The discussion above on bimodality holds for these file systems as well; FFS-like clustering policies should maintain mostly heated clusters and mostly unheated clusters.

4.2 SERO file system tamper evidence

The second part of the answer to the question we posed at the beginning of this section is provided by a number of proposals that hash disk blocks to provide tamper evidence.

So why does hashing help to protect the integrity of the data? Basically because it is easy to compute a hash from a group of disk blocks, while it is hard to find another set of disk blocks with the specific hash. We discuss two file systems for archival storage that use hashes extensively. The first builds an index structure from the leaves up, and the second builds the index from the root down.

Venti [40] uses a secure hash as the address of a node, where a node consists of a block of data or hashes. Venti builds a hierarchy of nodes from the leaves upwards by storing the hashes of the children of a node in the parent. The hash of the root node represents the entire hierarchy. As long as the hash of the root is stored securely, tampering can be detected. To check a node we use the hash of the node as its address, then re-compute the hash of the node, and finally compare the computed hash to the address. A computed hash that does not match the address of the node presents evidence of tampering.

A SERO device would be appropriate to keep the hash of a node secure. For simplicity, assume that the granularity of a node in the Venti hierarchy is a line. Then heating the line that represents a node is sufficient to calculate and store the hash of the line RO. The most relevant node to be heated is the root node, because this protects the entire hierarchy.

Venti lays different hierarchies on the data blocks to be able to record different snapshots of the file system (for example one for every working day). The same idea can also be used to construct hierarchies for different subsets of the data, such as the data accessible to different users, of different projects, etc. This would offer fine grained protection. However, the more nodes are heated, the more WORM space on the medium is reduced to RO space, thus resulting in a reduction in usability.

A fossilised index [57] builds a tree from the root downwards. To insert a new node in the tree we start at the root, visiting all nodes down to a leaf until a free slot is found in which the hash of the new node can be inserted. The hash of the node completely determines which slot in an existing node must be used, and what path to traverse. The tamper evidence guarantee of the fossilised index relies on the assumption that once all the slots of a node have been filled, the storage device en-

sures that the node becomes RO, for example by copying it to a WORM device.

A SERO device would provide appropriate support for a fossilised index as it makes copying the completed node to the WORM unnecessary. Again, assuming that a node fits in a line, a completely filled node is simply heated.

5 Security analysis

Triggered by large corporate scandals in the recent past, Hsu and Ong [19] propose the following threat model for secure storage. Assume that a powerful attacker (i.e., a disgruntled employee, or a dishonest CEO) regrets the existence of a certain stored record, and that he wishes history to be rewritten by tampering with the storage system so that it “forgets” the record. The attacker can do this either by overwriting or erasing the record, or by masking the existence of the record by overwriting or erasing the index. We assume that the attacker would not like to draw attention to his actions, for instance by removing or physically destroying the storage system or parts thereof, and that the attacker would like to cover his tracks.

In terms of the threat model of Hasan et al. [14], the *attacker capability* is that of a powerful insider wielding influence over systems and the personnel responsible for the systems. The attacker has root permission on all systems connected to the storage device. The *asset goal* is the integrity and availability of specific files that the attacker wishes to compromise. The *access entry point* is the whole system stack including direct access to the storage device. The attacker is expected to be able to disconnect the storage device temporarily from the system, then to connect it to a laptop with the appropriate interface for a limited period of time, and after he has finished to reconnect the device to the system.

This threat model represents a formidable challenge to the design of any secure storage system. For example some of the existing commercial WORM-based storage systems make it difficult to tamper with data, but on most systems tampering cannot be detected. We are not able to prevent tampering either, but we are able to detect tampering. We believe this to be a significant step towards addressing the challenge of secure storage.

WORM storage is geared towards providing integrity and availability. Therefore we will analyse to what extent our SERO system can cope with threats on integrity and availability. To ensure confidentiality or authenticity cryptographic techniques should be used, but this is beyond the scope of our paper.

5.1 Integrity

Assume that the attacker issues a write command, either indirectly via the file system (which is easy) or directly, to the device (which is harder) to alter a heated file. (Files that have not been heated are trivial to attack and are therefore beyond the scope of the security analysis.) Then there are four possibilities.

- mwb hash: Changing the magnetisation of an electrically written bit of the hash has no effect, as only the presence or the absence of a magnetic dot is relevant for a heated hash.
- mwb inode/data: Changing the magnetisation of a magnetically written bit of the data is detected by the verify operation as evidence of tampering.
- ewb hash: The only changes possible to an electrically written hash are $UH \rightarrow HH$ or $HU \rightarrow HH$. HH is an illegal code, and thus represents evidence of tampering.
- ewb inode/data: Data is read magnetically, so an electrically written bit in the data, which destroys the magnetic properties of the relevant dot, appears as a read error. However, a more subtle attack would be an attempt to split a file or to coalesce two files. To illustrate such an attack imagine a heated file as shown below, where the data block d_p is carefully crafted to look like a valid hash h' and where the data block d_{p+1} looks like a valid inode i' :

before:	$h \ i \ d_0 \ \dots \ d_{p-1} \ d_p \ d_{p+1} \ \dots \ d_q$
after:	$h \ i \ d_0 \ \dots \ d_{p-1} \ h' \ i' \ \dots \ d_q$

Assume that when instructed to heat the file with inode $d_{p+1} = i'$, the device has no way of telling whether this is a true file, or just part of the data of another file. Hence after heating, the original file with inode i would appear corrupted whereas the new file with inode i' appears to be genuine, quite the contrary of what we expect. Similarly, if the hashes are not in well-defined locations it is possible to coalesce two files making the result look genuine instead of the original. To prevent splitting or coalescing attacks, the device insists that hashes are written at known physical addresses.

In all four cases either the attempt to interfere with the integrity of the data is detected or the integrity is maintained.

5.2 Availability

As stated in the threat model, the device (or parts thereof) is not assumed to be taken off line for extended periods of time, or to be removed entirely. Therefore, the only way in which the availability of a file can be affected is when the access path to a heated file is blocked, or when another file masks the desired file.

Assume that the attacker tries to delete a heated file using the `rm` command. This removes the directory entry and tries to decrement the reference count in the inode. This implies writing the inode, which will be tamper-evident because the hash is invalidated. (Incidentally, it will not be possible to use the `ln` command on a heated file either, as this would increase the reference count in the inode.) A possible protection against malicious use of the `rm` command would be to maintain the directory as a fossilised index [57].

Assume that the attacker would like to create an exact copy of file to mask the existence of the original. This cannot be done since the physical addresses of the blocks are included in the calculation of the hash. Therefore, a copy can always be distinguished from an original.

Assume that the attacker clears the directory structure, then a `fsck` style scan of the medium would definitely recover (albeit slowly) all the heated files.

Assume that the attacker clears the entire medium, for example using a bulk eraser. If done properly [12], this would clear all magnetically written information. However all electrically written information is still present, thus providing the required evidence of tampering.

There are many attacks possible that our system cannot detect. For example assume that an attacker creates a new file with data that conflicts with the file the attacker wishes to remove. Firstly, the notion of conflicting data is a semantic notion that can only be resolved by the application. Secondly, this is not an attack on the integrity of the file per se (as the original remains untouched), but an attack on the authenticity. To prevent such attacks, cryptographic means are needed.

This concludes the preliminary security evaluation of the system, and also the speculative part of the paper. The next two sections describe the components of a SERO device and experimental evidence that heating is a feasible WO operation.

6 Probe storage on a patterned medium

A patterned medium needs an actuator, appropriate read/write heads, etc. to access the medium; a probe storage device would be highly suitable for this purpose. We discuss as an example the Twente Micro Scanning Probe Array Memory (μ SPAM) (Figure 4), which is made out of two or more silicon wafers bonded to each other. One

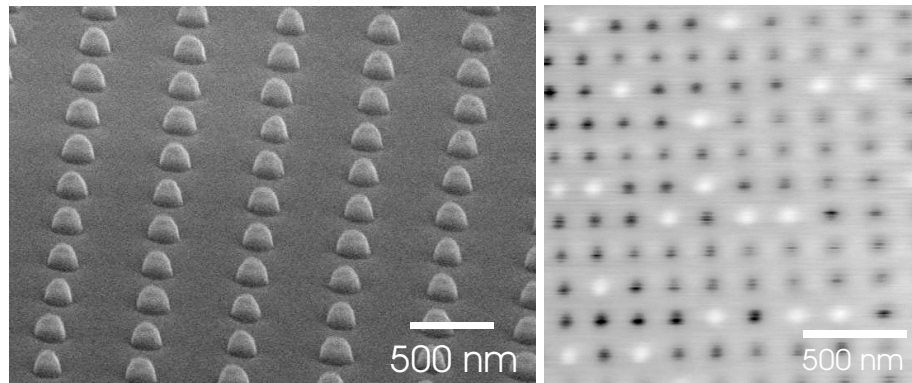


Figure 5: Scanning Electron Microscopy (left, 500 nm pitch) and Magnetic Force Microscopy image (right, 200 nm pitch) of two different patterned media. In the SEM image, the dots are still covered by a thick resist layer, and SEM images have a long depth of focus which gives the illusion that the dots are elliptic, whereas in reality they are not.

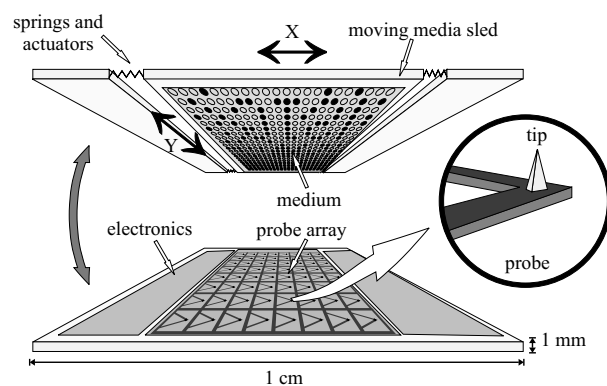


Figure 4: Principle of probe-based storage [39]. The system consists of two components, which are mounted on top of each other – one with the recording medium and the actuator, the other with the read/write probe array and the electronics.

half contains the (magnetic) medium. An electrostatic stepper actuator, such as the μ Walker [48] or Harmonica drive [43] is used to move the medium. The other half consists of one large array of probes.

The patterned medium The medium for the μ SPAM is a regular matrix of magnetic single domain dots. Such a discrete medium is expected to be able to support higher bit densities compared to the continuous polycrystalline medium used in the hard disk today [51].

A matrix with a period of 200 nm can be achieved [53]. A scanning electron microscope and magnetic image is shown in Figure 7. An improved setup with periodicities down to 150 nm has recently been realised [25], and a period of 100 nm (being 50 nm dot size and 50 nm spac-

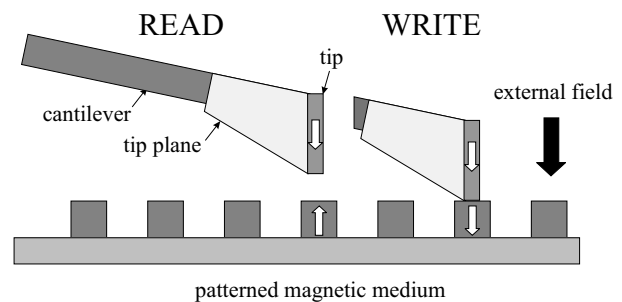


Figure 6: The principle of an MFM-measurement

ing) should be achievable. This will give a capacity of 10 Gbit/cm² (=65 Gbit/inch²).

The probes For reading, the μ SPAM uses the MFM (Magnetic Force Microscopy)-principle [38]. An MFM-probe is made by placing a small magnetic element, the tip, on a cantilever spring. Typical dimensions are a cantilever length of 200 μ m, element length of 4 μ m and diameter of 50 nm and a distance from the surface of 30 nm.

Figure 6 shows the principle of an MFM-measurement. The magnetic tip is attracted or repelled, depending on the stray field of the medium. The tip is affected by the magnetic orientation of a dot. The displacement of the cantilever might be measured by means of the change in capacity between the cantilever and the medium. Bits can be written by the combined field of the magnetic tip on one side of the medium and an externally applied field generated by a coil placed on the other side of the medium [53].

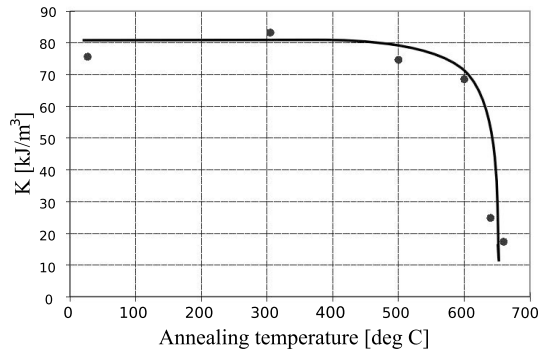


Figure 7: Perpendicular anisotropy as a function of the heating temperature

7 Heating changes magnetic properties

To support the heat operation, it should be possible to destroy the out-of-plane magnetic property of the dots. For this we need to discuss in more detail the internal structure of a dot and its relation to the magnetic properties.

The magnetic field energy is minimal when the magnetisation lies in the long axis of an object. The preferential direction of magnetisation in a needle is for instance along the needle, and not perpendicular to it. Since our dots are circular disks with a diameter much larger than the thickness, the magnetisation prefers to lie within the plane of the disk in the absence of any other energy contributions. The fact that the energy depends on the orientation of the magnetisation is called *anisotropy*. The preferred direction is called an *easy axis*. In the case of a dot which is perfectly circular, we speak of an easy plane. Normally dots will not be exactly circular, but elliptic. The magnetisation of the dot will prefer to lay in-plane along the long axis of the dot. By intentionally realising elliptic dots with their long axis along the track direction [32], data detection will be more robust and one can even imagine writing data into damaged dots (See the discussion of the erb operation in Section 3). Since the anisotropy is low, data density cannot be high however. In any case, magnetic dots with a diameter larger than their thickness will *a priori* have an in-plane easy axis.

For our system to work, we initially need dots with a perpendicular easy axis. Therefore a second strong energy contribution is needed to overcome the stray field energy and force the magnetisation perpendicular to the dot. This second energy term originates from the asymmetric arrangement of the atoms in the dot. In conventional perpendicular hard disk media, specific crystal structures are used to induce perpendicular anisotropy. These can however not easily be destroyed. In our case therefore we use interface properties. The magnetic ma-

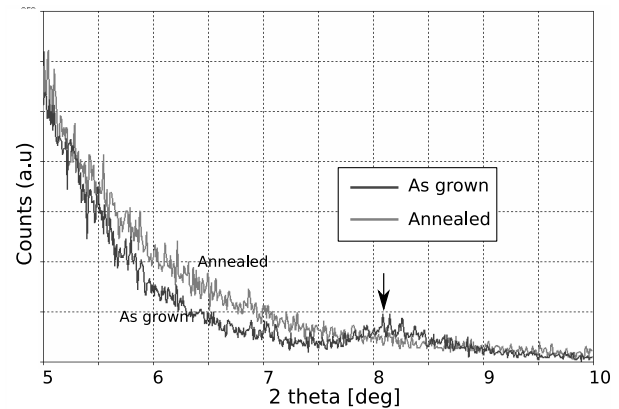


Figure 8: X-ray diffraction under a low angle of two samples, one with (labelled: Annealed) and one without annealing (labelled: As grown).

terial used in the dots consists of a stack of extremely thin Co and Pt layers, each no more than 1 nm thick [53]. The interfaces between the Co and Pt films cause anisotropy with the easy axis perpendicular to the interfaces.

Due to the delicate structure of these films, they do not support high temperatures over long periods of time. Above a certain temperature, the interface between the Co and Pt mixes, and the perpendicular interface anisotropy is destroyed. As a result the easy axis of magnetisation rotates back into the film plane. This is an irreversible process. After heat treatment, the interfaces cannot be restored.

To determine at which temperature interface mixing occurs, we have measured the anisotropy constant K of samples subjected to six different temperatures. The anisotropy constants were calculated by a Fourier transformation of the torque curve obtained with an applied field of 1350 kA/m. Figure 7 shows the dependence of the anisotropy value as a function of the heating, or annealing, temperature. (In materials science it is common to use the word annealing rather than heating, since it describes the process rather than the method.)

The perpendicular anisotropy of the unannealed film is 80 kJ/m^3 . This value is maintained up to an annealing temperature of $500 \text{ }^\circ\text{C}$. Above $600 \text{ }^\circ\text{C}$ the value of K drops dramatically. This means that for this particular film, heating temperatures over $500 \text{ }^\circ\text{C}$ will be required for permanent modification of the magnetic properties.

To investigate what happens to the interface between the Co and Pt in the films, we performed X-ray diffraction experiments. In this method the film is exposed to a non-destructive X-ray. The beam penetrates metals, and reflects from discontinuities such as atomic crystal planes or film interfaces. By varying the angle of inci-

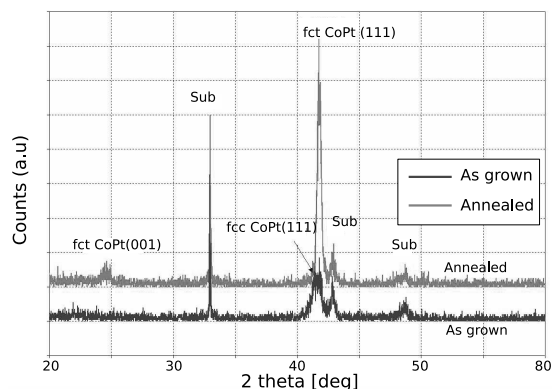


Figure 9: X-ray diffraction under a high angle of the same two samples as for low angle XRD, with annealing (labelled: Annealed) and without annealing (labelled: As grown).

dence, reflections from parallel planes at specific spacing add up and we observe a peak in the reflected intensity. In conventional operation (at high angles of incidence, so we observe spacings in the order of Angstroms), we can therefore determine the crystal structure of the film. At low angle of incidence we are sensitive to much larger spacing (1 nm) and we can observe the multilayer structure. The unannealed sample and a sample annealed at 700 °C were investigated by low angle X-ray diffraction (XRD) (Figure 8). A peak around 8 degrees on the 2θ axis is visible on the sample without annealing. This peak is due to the periodicity of the Co and Pt multilayers. From this angle, we can calculate that layer has a thickness of 0.6 nm. In the annealed sample, this peak has disappeared, which clearly shows that after an annealing treatment at a temperature higher than 600 °C, the interfaces have mixed, the perpendicular anisotropy is lost and the out-of-plane magnetic properties of the film are destroyed.

One might worry that by heating the interfaces are destroyed, but crystal structures are formed which induce perpendicular anisotropy. To study the change in crystal structure, we measured the samples by high-angle-XRD, so that we are sensitive to Angstrom spacing (Figure 9). In the annealed sample, we can find a strong reflection peak around 41.7 degrees in the 2θ axis. This peak can be characterized to a specific Co-Pt (111) crystal plane (face centered cubic, fcc). It suggests that indeed a new crystalline structure of fct Co-Pt was formed in the film. This crystal exhibits however magnetic anisotropy in the [001] direction, i.e., there are tilted magnetic easy axes in the film (not perpendicular, not in plane). So there is no risk that after excessive heating the perpendicular anisotropy can be restored by crystallisation.

We envisage that heating of the magnetic dots will be realised by passing a current from the probe tip to the dot. It has been shown in earlier work that these currents are even capable of evaporating the material, so the energy density is sufficiently high [36, 35]. This method is not only limited to probe storage however. Active research is being performed into hard disks with heat assisted writing strategies [28]. It is not fundamentally impossible that the supplied energy can be high enough to modify the magnetic properties of the disk permanently. In principle this method could therefore also be used in magnetic disk drives, although the implementation would be far more difficult.

More research will be needed to determine the time required, the amount of energy dissipated, the wear on the tip, and the effect of heating one dot on the neighbouring dots. Especially the last effect could be detrimental, since the magnetic state, or even the write-ability of the adjacent dot could be affected. However, it is not unlikely that by tailoring the materials and layer structures, the interface mixing temperature can be reduced, which will reduce the risk of thermal erasure of the neighbouring dots. Furthermore, by properly designing the thermal properties of the dot and the substrate, most of the heat can be conducted away into the substrate, rather than dissipating away laterally, like is done in magneto-optic medium [29]. In this way the heated area can be limited and damage to adjacent dots can be reduced. In any case it will be necessary to use the write-once operation sparingly.

8 Discussion

We have described an experiment in material science and discussed a number of questions about how probe-based storage on a patterned medium can be used to build a tamper-evident storage device and file system. The experiments and discussion raise many more issues that must be addressed in future work. We describe the most relevant questions below.

Efficiency The storage efficiency of the system merits some discussion. Firstly, we have explained the low level system operations using a simple Manchester encoding for the hash. For large N the amount of space wasted is negligible (1 block out of 2^N), but the price to pay is lack of flexibility. For small values of N we could employ more efficient coding techniques [33].

Secondly, the storage system as we have described it behaves as mass storage that can be read and written any number of times as one would expect, except that once an area has been heated, it can no longer be rewritten with impunity. This means that over the lifetime of the device,

the read/write area gradually shrinks, and the read-only area grows, until the device has become a pure read-only device. The medium can safely be decommissioned by the time all data has expired. This means that the lifetime of the data must be matched to the lifetime of the medium.

Deletion Once heated, data will remain until the medium is decommissioned. This is not desirable if there is a large variation in the lifetime of the data, particularly in cases where retention periods are carefully controlled by regulation. There are several ways to deal with this problem. Firstly, data could be written encrypted, disposing of the encryption key as soon as the expiry date of the data is reached [8]. Secondly, it is possible to implement a physical shred operation on the device (similar to what has been achieved for optical storage [45]), which in our case would physically destroy the expired data by precise local heating. However, both approaches are vulnerable to attacks by a dishonest CEO and as such not wholly satisfactory. We would advocate data to be segregated by expiry date, thus making it possible to take a device physically out of service. Given the enormous volume of data subject to compliance regulation [54] this should be possible to arrange.

Forensics A typical server is responsible for so much data that the traditional disk imaging approach, which copies an entire disk at the lowest level possible (i.e., including unused and bad blocks), is becoming infeasible. Firstly the volume of data may be prohibitively large, and secondly, to image the disks the server must be stopped, possibly for hours, thus losing valuable production time. Live forensics methods [1] would benefit from a storage device that can be instructed to heat evidence without having to copy it. One of the most difficult problems in this field is to speed up the collection of evidence [9] in a kind of digital evidence bag. Our heated files could be the basis of such an evidence bag. It should be kept in mind that a forensic investigation carried out by the police is relatively rare, and probably still requires whole disk imaging. A forensic investigation by company staff is more common, since companies will try to deal with the problems such as harassment, and theft in house. Problems such as child pornography and money laundering must always be reported to the police [13].

We are confident that even a skilled focused ion beam (FIB) operator would find it difficult to reconstruct a perfect out-of-plane dot because she would have to remove the debris of an in-plane dot first, and then deposit several thin Co and Pt layers in a sub-micron area with the correct delicate layer structure to obtain perpendicular anisotropy, just to reconstruct one dot. Using magnetic

imaging techniques [27], a forensics team would probably have no difficulty identifying a reconstructed out-of-plane dot from an original out-of-plane dot.

Tamper-evident storage as a building block Our system offers tamper-evident storage, which could be used as a building block in other systems. For example the idea of self-securing storage [47] takes the view that the storage system should place only limited trust in the host that controls it, since the host is more likely to become compromised than the storage system. Thus the storage system itself maintains a log of the instructions it is given, and ensures that earlier versions of any file (within a given time window) can be recovered. Our approach could strengthen the defences of a self-securing storage device because the logs can be heated.

9 Conclusions and future work

Probe storage on patterned media is a promising technology for developing tamper-evident storage. The capacity of such devices will be huge, and the tamper evidence is good. The measurements reported in the paper demonstrate that in principle it is possible to use a patterned magnetic medium in two essentially different ways: for normal read-write purposes and for read-only purposes after the data has been heated. It is physically impossible to alter the data without being detected after the heat operation has been used. We discuss the main issues that must be addressed when designing a device and a file system for tamper-evident SERO storage. The strong point of the SERO approach is its combination of the advantages of WORM storage with the advantages of WMRM storage.

We have identified the most relevant issues in the design of the system. However, much work remains to be done. On the software side we plan to design and build a simulation of the device and the file system, such that we can study the performance/security tradeoffs. The next step would be to develop a time-accurate emulator for the device, as well as an implementation of the file system to validate the simulation results. The time-accurate emulator could probably be built using anti-fuse based write once semiconductor memory technology as used in FPGAs. On the hardware side we plan to develop materials that change magnetic properties by interface mixing at lower temperatures, and tips that generate enough heat for interface mixing, studying the efficiency and reliability of the mechanisms involved.

10 Acknowledgements

We thank Sebastiaan Konings, Rogelio Murillo, and Takahiro Onoue for their help with the measurements. Jeroen Doumen, Sape Mullender, and Berend-Jan van der Zwaag provided helpful comments on the paper. The efforts of our shepherd Petros Maniatis are gratefully acknowledged.

References

- [1] ADELSTEIN, F. Live forensics: diagnosing your system without killing it first. *Commun. ACM* 49, 2 (Feb 2006), 63–66.
- [2] ALON, N., KAPLAN, H., KRIVELEVICH, M., MALKHI, D., AND STERN, J. Scalable secure storage when half the system is faulty. *Information and Computation* 174, 2 (May 2002), 203–213.
- [3] ANDERSON, R. J., AND KUHN, M. G. Tamper resistance - A cautionary note. In *2nd Int. Usenix Workshop on Electronic Commerce* (Oakland, California, Nov 1996), USENIX Association, pp. 1–11.
- [4] ANDERSON, R. J., AND KUHN, M. G. Low cost attacks on tamper resistant devices. In *Security protocols: 5th Int. Workshop* (Paris, France, Apr 1997), M. Lomas and B. Christianson, Eds., vol. LNCS 1361, Springer, pp. 125–136.
- [5] APVRILLE, A., HUGHES, J., AND GIRIER, V. Streamed or detached triple integrity for a time stamped secure storage system. In *1st Int. IEEE Security in Storage Workshop (SiSW)* (Greenbelt, Maryland, Dec 2002), IEEE Computer Society, pp. 53–64.
- [6] BALDWIN, A., AND SHIU, S. Enabling shared audit data. *International Journal of Information Security* 4, 4 (Oct 2005), 263–276.
- [7] BOBETH, M., HECKER, M., POMPE, W., SCHNEIDER, C. M., THOMAS, J., ULLRICH, A., AND WETZIG, K. Thermal stability of nanoscale Co/Cu multilayers. *Zeitschrift fuer Metallkunde/Materials Research and Advanced Techniques* 92, 7 (2001), 810–819.
- [8] BONEH, D., AND LIPTON, R. J. A revocable backup system. In *6th USENIX Security Symp. Focusing on Applications of Cryptography* (San Jose, California, Jul 1996), USENIX Association, pp. 91–96.
- [9] CASEY, E. Investigating sophisticated security breaches. *Commun. ACM* 49, 2 (Feb 2006), 48–55.
- [10] GENDRON, Y. Reforming auditor independence: Voicing and acting upon auditors’ concerns and criticisms. *Advances in Public Interest Accounting* 12 (2006), 103–118.
- [11] GORDON, L. A., LOEB, M. P., LUCYSHYN, W., AND RICHARDSON, R. *10th Annual CSI/FBI Computer crime and security survey*. Computer Security Institute, San Francisco, California, 2005.
- [12] GUTMANN, P. Secure deletion of data from magnetic and solid-state memory. In *6th USENIX Security Symp.* (San Jose, California, Jul 1996), USENIX Association, pp. 77–89.
- [13] HAGGERTY, J., AND TAYLOR, M. Managing corporate computer forensics. *Computer Fraud & Security* 2006, 6 (Jun 2006), 14–16.
- [14] HASAN, R., MYAGMAR, S., LEE, A. J., AND YURCIK, W. Toward a threat model for storage systems. In *1st ACM Workshop on Storage Security and Survivability (StorageSS)* (Fairfax, Virginia, Nov 2005), ACM, pp. 94–102.
- [15] HASAN, R., TUCEK, J., STANTON, P., YURCIK, W., BRUMBAUGH, L., ROSENDALE, J., AND BOONSTRA, R. The techniques and challenges of immutable storage with applications in multimedia. In *Storage and Retrieval Methods and Applications for Multimedia*, R. W. Lienhart, N. Babaguchi, and E. Y. Chang, Eds., vol. 5682. SPIE, Jan 2005, pp. 41–52.
- [16] HASAN, R., AND YURCIK, W. A statistical analysis of disclosed storage security breaches. In *2nd ACM Workshop on Storage Security and Survivability (StorageSS)* (Alexandria, Virginia, Oct 2006), ACM, pp. 1–8.
- [17] HASAN, R., YURCIK, W., AND MYAGMAR, S. The evolution of storage service providers: techniques and challenges to outsourcing storage. In *1st ACM Workshop on Storage Security and Survivability (StorageSS)* (Fairfax, Virginia, Nov 2005), ACM, pp. 1–8.
- [18] HONG, B., WANG, F., BRANDT, S. A., LONG, D. D. E., AND SCHWARZ, T. J. E. Using MEMS-based storage in computer systems—MEMS storage architectures. *Trans. Storage* 2, 1 (Feb 2006), 1–21.
- [19] HSU, W. W., AND ONG, S. WORM storage is not enough. *IBM Systems Journal* 46, 2 (Apr 2007), 363–372.
- [20] HURLEY, J. The CSO’s security compliance agenda: Benchmark research report. *Computer Security Journal* 22, 1 (Dec 2006), 37–44.
- [21] JAQUETTE, G. A. Tamper resistant write once recording of a data storage cartridge having rewritable media. International Business Machines Corporation (Armonk, NY), Mar 2007. Patent Nr. 7,193,803.
- [22] KIM, Y.-S., JANG, S., LEE, C. S., JIN, W.-H., CHO, I.-J., HA, M.-H., NAM, H.-J., BU, J.-U., CHANG, S.-I., AND YOON, E. Thermo-piezoelectric Si_3N_4 cantilever array on CMOS circuit for high density probe-based data storage. *Sensors and Actuators, A: Physical* 135, 1 (Mar 2007), 67–72.
- [23] KURTAS, E. M., ERDEN, M. F., AND YANG, X. Future read channel technologies and challenges for high density data storage applications. In *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)* (Philadelphia, Pennsylvania, Mar 2005), IEEE, pp. V737–V740.
- [24] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *6th Symp. on Operating Systems Design and Implementation (OSDI)* (San Francisco, California, 2004), USENIX Association, pp. 9–9.
- [25] LUTTGE, R., VAN WOLFEREN, H. A. G. M., AND ABELMANN, L. Nanolithography for patterned magnetic data storage media. *Journal of Vacuum Technology B* (2007), accepted for publication.
- [26] MANIATIS, P., AND BAKER, M. Secure history preservation through timeline entanglement. In *11th USENIX Security Symp.* (San Francisco, California, Aug 2002), USENIX Association, pp. 297–312.
- [27] MAYERGOYZA, I. D., SERPICO, C., KRAFFT, C., AND TSE, C. Magnetic imaging on a spin-stand. *J. of Applied Physics* 87, 9 (May 2000), 6824–6826.
- [28] MCDANIEL, T. W., CHALLENGER, W. A., AND SENDUR, K. Issues in heat-assisted perpendicular recording. *IEEE Transactions on Magnetics* 39, 4 (2003), 1972–1979.
- [29] MCDANIEL, T. W., AND SEQUEDA, F. O. Design material selection for a thin film magneto-optic disk. *Applied physics communications* 11, 4 (1992), 427–445.
- [30] MIN, D.-K., AND HONG, S. Design and analysis of the position detection algorithm for a probe storage. *IEEE Sensors Journal* 6, 4 (Aug 2006), 1010–1015.

- [31] MOLNAR, D., KOHNO, T., SASTRY, N., AND WAGNER, D. Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- how to store ballots on a voting machine (extended abstract). In *IEEE Symp. on Security and Privacy (S&P)* (Berkeley, California, May 2006), IEEE Computer Society, pp. 365–370.
- [32] MORALEJO, S., NO, F. J. C., REDONDO, C., JI, R., NIELSCH, K., ROSS, C. A., AND NO, F. C. Fabrication and magnetic properties of hexagonal arrays of NiFe elongated nanomagnets. *Journal of Magnetism and Magnetic Materials* 316, 2 (Sep 2007), e44–e47.
- [33] MORAN, T., NAOR, M., AND SEGEV, G. Deterministic History-Independent strategies for storing information on Write-Once memories. In *34th Int. Colloquium on Automata, Languages and Programming (ICALP)* (Wroclaw, Poland, Jul 2007), vol. LNCS 4596, Springer, pp. 305–315.
- [34] NABERHUIS, S. Probe-based recording technology. *J. of Magnetism and Magnetic Materials* 249, 3 (Sep 2002), 447–451.
- [35] ONOUE, T., SIEKMAN, M., ABELMANN, L., AND LODDER, J. C. Heat assisted magnetic probe recording onto a thin film with perpendicular magnetic anisotropy. *Journal of Applied Physics D* (2007), accepted for publication.
- [36] ONOUE, T., SIEKMAN, M. H., ABELMANN, L., AND LODDER, J. C. Probe recording on CoNi/Pt multilayered thin films by using an MFM tip. *Journal of Magnetism and Magnetic Materials* 272-276, III (May 2004), 2317–2318.
- [37] PATZAKIS, J. New accounting reform laws push for Technology-Based document retention practices. *Int. Journal of Digital Evidence* 2, 1 (Spring 2003), paper 2.
- [38] PORTHUN, S., ABELMANN, L., AND LODDER, C. Magnetic force microscopy of thin film media for high density magnetic recording. *J. of Magnetism and Magnetic Materials* 182, 1-2 (Feb 1998), 238–273.
- [39] POZIDIS, H., BÄCHTOLD, P., BONAN, J., CHERUBINI, G., ELEFThERIOU, E., DESPONT, M., DRECHSLER, U., DÜRIG, U., GOTSMANN, B., HÄBERLE, W., HAGLEITNER, C., JUBIN, D., KNOLL, A., LANTZ, M. A., PANTAZI, A., ROTHUIZEN, H. E., SEBASTIAN, A., STUTZ, R., AND WIESMANN, D. W. Scanning probes entering data storage: From promise to reality. In *IEEE Conf. on Emerging Technologies - Nanoelectronics* (Singapore, Jan 2006), IEEE, pp. 39–44.
- [40] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival data storage. In *1st USENIX Conf. on File and Storage Technologies (FAST)* (Monterey, California, Jan 2002), USENIX Association, pp. 89–101.
- [41] RETTNER, C. T., ANDERS, S., BAGLIN, J. E. E., THOMSON, T., AND TERRIS, B. D. Characterization of the magnetic modification of Co/Pt multilayer films by He⁺, Ar⁺, and Ga⁺ ion irradiation. *Applied Physics Letters* 80, 2 (Jan 2002), 279–281.
- [42] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb 1992), 26–52.
- [43] SARAJLIC, E., BERENSCHOT, E., TAS, N. R., FUJITA, H., KRIJNEN, G., AND ELWENSPOEK, M. C. Fabrication and characterization of an electrostatic contraction beams micromotor. In *IEEE Int. Conf. on Micro Electro Mechanical Systems (MEMS)* (Istanbul, Turkey, Jan 2006), IEEE, pp. 814–817.
- [44] SELTZER, M. I., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. N. File system logging versus clustering: A performance comparison. In *Technical Conf. on UNIX and Advanced Computing Systems* (New Orleans, Louisiana, Jan 1995), USENIX Association, pp. 249–264.
- [45] SKEETER, B. J., WORBY, B. L., HOLSTINE, K. R., AND BOLT, D. A. Optical disk shred operation with detection. United States Patent and Trademark Office, Mar 2007. Patent Application Nr. 20070047395.
- [46] SPOERL, K., AND WELLER, D. Interface anisotropy and chemistry of magnetic multilayers: Au/Co, Pt/Co and Pd/Co. *Journal of Magnetism and Magnetic Materials* 93 (Feb 1991), 379–385.
- [47] STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A., AND GANGER, G. R. Self-Securing storage: Protecting data in compromised systems. In *Foundations of Intrusion Tolerant Systems (OASIS)*. IEEE Computer Society, 2003, pp. 195–209.
- [48] TAS, N. R., WISSINK, J., SANDER, A. F. M., LAMMERINK, T. S. J., AND ELWENSPOEK, M. C. Modeling, design and testing of the electrostatic shuffle motor. *Sensors and Actuators A (Physical)* 70, 1-2 (Oct 1998), 171–178.
- [49] TAYLOR, M. The EU data retention directive. *Computer Law & Security Report* 22, 4 (2006), 309–312.
- [50] TERRIS, B. D., FOLKS, L., WELLER, D., BAGLIN, J. E. E., KELLOCK, A. J., ROTHUIZEN, H., AND VETTIGER, P. Ion-beam patterning of magnetic films using stencil masks. *Applied Physics Letters* 75, 3 (Jul 1999), 403–405.
- [51] TERRIS, B. D., AND THOMSON, T. Nanofabricated and self-assembled magnetic structures as data storage media. *J. of Physics D: Applied Physics* 38, 12 (Jun 2005), R199–R222.
- [52] TERRIS, B. D., THOMSON, T., AND HU, G. Patterend media for future magnetic data storage. *Microsystem Technologies* 13, 2 (Jan 2007), 189–196.
- [53] VALLEJO, R. M., SIEKMAN, M. H., BOLHUIS, T., ABELMANN, L., AND LODDER, J. C. Thermal stability and switching field distribution of CoNi/Pt patterned media. *Microsystem Technologies* 13, 2 (Jan 2007), 177–180.
- [54] VAN WANROOIJ, W., AND PRAS, A. Data on retention. In *16th IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management (DSOM)* (Barcelona, Spain, Oct 2005), vol. LNCS 3775, Springer, pp. 60–71.
- [55] WANG, Y., AND ZHENG, Y. Fast and secure magnetic WORM storage systems. In *2nd IEEE Int. Security in Storage Workshop (SISW)* (Washington, DC, Oct 2003), IEEE Computer Society, pp. 11–11.
- [56] WINARSKI, C. J., AND DIMITRI, K. E. Write-once read-many hard disk drive. International Business Machines Corporation (Armonk, NY), Apr 2005. Patent Nr. 6,879,454.
- [57] ZHU, Q., AND HSU, W. W. Fossilized index: the linchpin of trustworthy non-alterable electronic records. In *ACM Int. Conf. on Management of Data (SIGMOD)* (Baltimore, Maryland, Jun 2005), ACM, pp. 395–406.

SWEEPER: An Efficient Disaster Recovery Point Identification Mechanism

Akshat Verma
IBM India Research
akshatverma@in.ibm.com

Kaladhar Voruganti
Network Appliance
kaladhar.voruganti@netapp.com

Ramani Routray
IBM Almaden Research
routrayr@us.ibm.com

Rohit Jain¹
Yahoo India
rohitj@yahoo-inc.com

Abstract

Data corruption is one of the key problems that is on top of the radar screen of most CIOs. Continuous Data Protection (CDP) technologies help enterprises deal with data corruption by maintaining multiple versions of data and facilitating recovery by allowing an administrator restore to an earlier clean version of data. The aim of the recovery process after data corruption is to quickly traverse through the backup copies (old versions), and retrieve a clean copy of data. Currently, data recovery is an ad-hoc, time consuming and frustrating process with sequential brute force approaches, where recovery time is proportional to the number of backup copies examined and the time to check a backup copy for data corruption.

In this paper, we present the design and implementation of *SWEEPER* architecture and backup copy selection algorithms that specifically tackle the problem of quickly and systematically identifying a good recovery point. We monitor various system events and generate checkpoint records that help in quickly identifying a clean backup copy. The *SWEEPER* methodology dynamically determines the selection algorithm based on user specified recovery time and recovery point objectives, and thus, allows system administrators to perform trade-offs between recovery time and data currentness. We have implemented our solution as part of a popular Storage Resource Manager product and evaluated *SWEEPER* under many diverse settings. Our study clearly establishes the effectiveness of *SWEEPER* as a robust strategy to significantly reduce recovery time.

1 Introduction

Data Resiliency is a very important concern for most organizations to ensure business continuity in the face of different types of failures and disasters, such as virus attacks, site failures, machine/firmware malfunction, accidental and malicious human/application errors [17, 2].

Resiliency is not only about being able to resurrect data after failures, but also about how quickly the data can be resurrected so that the business can be operational again. While in the case of total data loss failures, the recovery time is largely dominated by *Restoration Cost*, i.e., time to restore the data from backup systems at local or remote locations; in the case of data corruption failures, the time to identify a clean previous copy of data to revert to can be much larger. Often, data corruption is detected after the incidence of corruption itself. As a result, the administrator has a large number of candidate backup copies to select the recovery point from. The key to fast recovery in such cases is reducing the time required in the identification step. Much research and industrial attention have been devoted to protecting data from disasters. However, relatively little work has been done in the area of how to quickly retrieve the latest clean copy of uncorrupted data. The focus of this paper is on how to efficiently identify clean data.

Data resiliency is based on *Data Protection*: taking either continuous or periodic snapshots of the data as it is being updated. Block level [7] [15], file level [10], logical volume level [27] and database level [5] data replication/recovery mechanisms are the most prominent data protection mechanisms. The mechanisms vary with respect to their support for different data granularities, transactional support, replication site distance, backup latencies, recovery point and recovery time objectives [11]. Continuous data protection (CDP) [24] is a form of continuous data protection that allows one to go back in time after a failure and recover earlier versions of an object at the granularity of a single update.

The recovery process is preceded by *Error Detection*. Errors are usually found either by application users or by automated data integrity tools. Tools such as disk scrubbers and S.M.A.R.T. tools [23] can detect corruptions caused by hardware errors. Virus scanners, application specific data integrity checkers such as fsck for filesystem integrity, and storage level intrusion detection

tools [19, 3] can detect logical data corruptions caused by malicious software, improper system shutdown, and erroneous administrator actions. For complex Storage Area Network (SAN) environments, configuration validation tools [1] have been proposed that can be used in both proactive and reactive mode to identify configuration settings that could potentially lead to logical data corruptions.

Once system administrators are notified of a corruption, they need to solve the *Recovery Point Identification* problem, i.e., determine a recovery point that will provide a clean copy of their data. Typically, system administrators choose a recovery point by trading-off recovery speed (the number of versions that need to be checked to find a clean copy) versus data currentness (one might not want to lose valid data updates for the sake of fast recovery). Recovery point identification is currently a manual, error-prone and frustrating process for system administrators, due to the pressure to quickly bring organization's applications back on-line. Even though CDP technologies provide users with the ability to rollback every data update, they do not address the problem of identifying a clean copy of data. It is only after a good recovery point has been identified, that *Data Recovery* can begin by replacing the corrupt copy by the clean copy of data. The efficacy of a recovery process is characterized by a Recovery Time Objective (RTO) and a Recovery Point objective (RPO). RTO measures the downtime after detection of corruption, whereas RPO indicates the loss in data currency in terms of seconds of updates that are lost.

This paper describes and evaluates methods for efficient recovery point identification in CDP logs which reduce RTO, while not compromising on RPO. The basic idea behind our approach is to evaluate the events generated by various components such as applications, file systems, databases and other hardware/software resources and generate checkpoint records. Subsequently upon the detection of failure, we efficiently process these checkpoint records to start the recovery process from an appropriate CDP record. Since CDP mechanisms are typically used along with point-in-time snapshot (PIT) technologies, it is possible to create data copies selectively. Further, since the time it takes to test a copy of data for corruption dominates overall recovery time, selective testing of copies can drastically reduce recovery time. This selective identification of copies that one can target for quick recovery is the focus of this work.

Some existing CDP solutions [16, 21, 30] are based on the similar idea of checkpointing *interesting* events in CDP logs. While event checkpointing mechanisms help in narrowing down the search space, they do not guarantee that the most appropriate checkpoint record will be identified. Thus, the CDP log evaluation techniques pre-

sented in this paper compliment these checkpoint record generation mechanisms in quickly identifying the most suitable CDP record. The key *contributions* of this paper are:

- ***SWEEPER* Recovery Point Identification Architecture:** We present the architecture of an extensible recovery point identification tool that consists of event monitoring, checkpoint generation, clean copy detection and CDP log processing components. The architectural framework is independent of specific applications and can easily be used with other application specific CDP solutions such as [16, 21].
- **Novel Event Checkpointing Mechanism:** Data corruptions are usually not silent but are accompanied by alerts and warning messages from applications, file systems, operating systems, SAN elements (e.g., switches, virtualization engines), storage controllers and disks. Table 2 lists some events that usually accompany data corruption caused by various components. We define a mechanism that identifies events from various application and system activities and uses a combination of a) expert provided knowledge base, b) resource dependency analysis, and c) a event correlation technique to correlate them with various types of corruption.
- **CDP Record Scanning Algorithms:** We present three different CDP record scanning algorithms that efficiently process the event checkpoints for identifying the appropriate CDP log record for data recovery. The scanning mechanisms isolate a recovery point quickly by using the observations that (a) pruning the space of timestamps into equal sized partitions reduces the search space exponentially and (b) checkpoint records that have high correlation with corruptions are more likely to be indicative of corruption. One of the novel features of the checkpoint record selection process is the acceptance of recovery time objective (RTO) and recovery point objective (RPO) as input parameters. Thus, the algorithms have the desirable property of providing a tradeoff between the total execution time and the data currency at the recovery point. The expected execution time of the algorithms is logarithmic in the number of checkpoint records examined and linear in the number of data versions tested for integrity. The proposed algorithms are robust with noise in the checkpoint record generation process, and can deal with errors in correlation probability estimation. Further, even though they are not designed to deal with silent errors, they still find a recovery point in logarithmic time for silent errors.

- **Performance Evaluation:** We present an implementation of the *SWEEPER* architecture and algorithms in context of a popular Storage Resource Manager product (IBM Total Storage Productivity Center), and demonstrate the scalability of our design. We also present a comparative analysis of the various Record Scanning algorithms proposed by us under different operational parameters, which include failure correlation probability distribution for events, number of checkpoint records, and rate of false negatives. We identify the different scenarios for which each algorithm is most suited and conclusively establish the efficacy of the *SWEEPER* methodology.

The rest of this paper is organized as follows. We formally define the *Recovery Point Identification* problem and model in Sec. 2.1 and 2.2. The *SWEEPER* architecture is presented in Sec. 2.3. The CDP record scanning algorithms are described in Sec.3, and our implementation in Sec. 4. Sec. 5 describes our experimental evaluation. Sec. 6 discusses the strengths and weaknesses of *SWEEPER* in the context of related work followed by a conclusion in Sec. 7.

2 Recovery-point Identification: Model and Architecture

We now provide a formal definition of the *Recovery Point Identification* problem and model parameters.

2.1 Problem Formulation

We investigate the problem of data recovery after a failure has resulted in data corruption. Data corruption is detected by an integrity checker (possibly with application support). The second step in this process is to identify the nature of corruption (e.g., virus corruption, hardware error). For isolating the problem, the components (e.g, controllers, disks, switches, applications) that may be the cause of error are identified by constructing a mapping from the corrupted data to the applications. Once the affected components are identified, the recovery module finds a time instance to go back to when the data was uncorrupted. Once the time instance is identified, CDP logs and point-in-time images are used to construct the uncorrupted data.

We now formally describe the *Recovery Point Identification* problem. The notations used in this paper are tabulated in Table. 1. The *Recovery Point Identification* problem is to minimize the total time taken to recover the data in order to get the most current uncorrupted data (i.e., find a timestamp T_i such that $T_i = T_e$ while minimizing D_{rec}). A constrained version of the problem is to minimize $T_e - T_i$ subject to a bound on D_{rec} . The

T_0	Time at which data was last known to be clean
T_d	Time at which corruption was detected
T_i	Timestamp i
T_e	Error Timestamp
T_p	Number of CDP logs after which a PIT copy is taken
N	Number of CDP logs between T_0 and T_d
N_c	Number of checkpoints between T_0 and T_d
D_{rec}	Time taken for recovery
C_p	Cost of getting a PIT copy online and ready to read/write
C_l	Average Cost of applying one CDP log entry
C_t	Cost of testing one data image for corruption
$p_{i,j}$	probability that checkpoint j is correlated with corruption i

Table 1: Notations

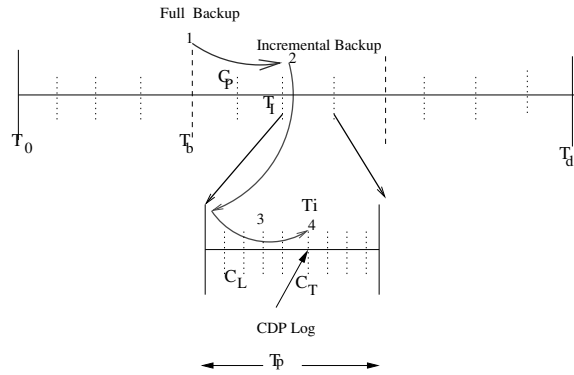


Figure 1: Timeline for testing the snapshot at time T_i : 1 Getting a full backup online. 2 Applying incremental backups over it. 3 Applying CDP logs to create the snapshot at time T_i . 4 Testing the data for integrity.

identification of T_i proceeds by finding an ordered set S of timestamps, which is a subset of the set of all the timestamps (T_0, \dots, T_N) such that S_m ($m = |S|$), the last element of the set S , is the same as the error point T_e . Further, the total cost of creating and testing the data images corresponding to the m timestamps in S , which equals D_{rec} , should be minimized. The cost of checking a data image at timestamp T_i for corruption is the sum of (a) the cost of making the first PIT image preceding the timestamp available for read/write (C_p) (b) the cost of applying the $T_i \% T_p$ (T_i modulo T_p) CDP logs ($C_l(T_i \% T_p)$) and (c) the cost of testing the copy (C_t). Hence, the total time spent in isolating the uncorrupted copy is the sum of the costs of all the timestamps checked in the sequence S .

2.2 Recovery Point Parameters and Estimating Sequential Checking Time

We now elaborate on the recovery parameters like C_p , C_t etc and their typical values. For a typical example, consider Fig. 1, where a *Recovery Point Identification* strategy decided to check data image at T_i for corruption. The data protection (continuous and point-in-time) solution employed in the setting takes a total backup of the data at regular intervals. In between total backups, incremental backups are taken after every T_p writes (CDP logs). The number of incremental backups taken between two consecutive total backups is denoted by f_I . Hence, in order to construct the point-in-time snapshot of data at time T_i , we make the first total backup copy preceding T_i (labeled as T_b in the example) online. Then, we apply incremental backups over this data till we reach the timestamp T_I of the last incremental backup point before T_i . The total time taken in getting the PIT copy at T_I online is denoted by C_p . On this PIT copy, we now apply the CDP logs that capture all the data changes made between T_I and T_i , and incur a cost of C_l for each log. Finally, an integrity check is applied over this data, and the running time of the integrity checker is denoted by C_t .

For average metadata and data write sizes of $E(S_m)$ and $E(S_w)$ respectively, write coalescing factor W_{cf} , average filesize $E(S_f)$, read and write bandwidth of B_r and B_w respectively, a file corpus of N_f files, and a unit integrity test time I_t , the expected time taken for each of these three activities are given by the following equations. (C_p calculation is based on the assumption that constructing the PIT copy only requires metadata updates.)

$$C_p = \frac{(f_I/2)(T_p/W_{cf})E(S_m)}{B_w},$$

$$C_l(T_i \% T_p) = \frac{(T_p/2)E(S_w)}{B_w} \quad (1)$$

$$C_t = \frac{N_f E(S_f)}{B_r} + N_f E(S_f) I_t \quad (2)$$

In a typical setup with 1000 files being modified every second, update sizes of 4KB, file sizes of 100KB, metadata size of 64Bytes, total backups every 12 hours, incremental backups every 1hr, and disk transfer rate of 100Mbps, the time taken to get a PIT image online C_p is approximately 50 seconds. The time taken to apply the CDP logs is of the order of 10 minutes, whereas the I/O time taken (not including any computations) by the integrity checker C_t comes to about 4 hours assuming that the integrity checker only needs to check the modified files. In a deployment with a CDP log window of 24 hrs, if one sequentially checks every 1000th record, the expected time to find the point of corruption would be

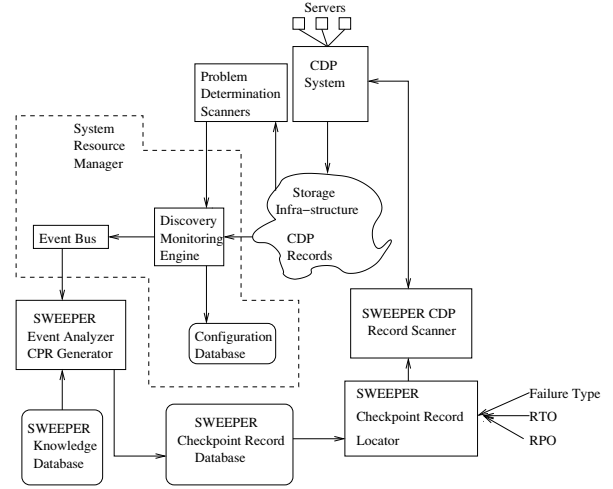


Figure 2: SWEEPER Recovery Point Identification Architecture

of the order of 100 days. Hence, sequential checking is infeasible and minimizing the time taken by the recovery process essentially boils down to minimizing the number of data images that are checked by recovery process.

2.3 SWEEPER Recovery Point Identification Architecture

The central *idea* of *SWEEPER* is to automatically checkpoint important system events from various application-independent monitoring sources as indicators of corruption failures. In contrast to earlier work [16] that requires an administrator to manually define the events for each application that are correlated with various corruption types, we automate the process of creating these checkpoints and indexing them with the type and scope of corruption along with a number that indicates the probability of the event being correlated with the specific corruption type. We then use these checkpoints as hints to quickly pinpoint the most recent clean copy of data. The overall architecture of *SWEEPER* is described in Fig. 2

The key design question while architecting a recovery mechanism like *SWEEPER* is whether to build it a) as part of a CDP system, or b) as part of a Storage Resource Manager product like EMC Control Center, HP OpenView or IBM TotalStorage Productivity Centre, or c) as a standalone component. We have separated the design of algorithms and the architecture so that the algorithms can work with any architecture and vice versa. Implementing *SWEEPER* outside the CDP system allows it to be leveraged by many different types of CDP systems. The disadvantage of this implementation is that some CDP systems might not completely expose all of the internal system state changes via an event mechanism. Most of this information is available from Stor-

age Resource Managers. Furthermore, most Storage Resource Managers have a comprehensive monitoring and resource discovery mechanism that can be leveraged by the Recovery module. Hence, we recommend the design choice (b) for *SWEEPER* (Fig. 2), with the following key components.

- **CDP System:** CDP system is essentially a data versioning system that can generate either log records or snapshots during data updates. The Users can retrieve snapshots using temporal queries based on timestamps, or they can simply traverse the snapshots using a cursor returned by query. The CDP system also allows for restoring the data in a previous copy and make it the latest copy.
- **Storage Infra-Structure:** The storage infra-structure consists of servers (hosts), switches, storage controllers (contain disks), file systems, database systems, virtualization boxes, and security boxes. A CDP system is also part of the storage infra-structure but we are showing it separately in Fig. 2.
- **Storage Resource Manager:** Storage resource managers contain a discovery/monitoring engine for monitoring storage infra-structure using a combination of SNMP protocol, CIM/SMI-S protocol, proprietary agents, in-band protocol discovery/monitoring agents, operating system event registries, proc file systems and application generated events. Event bus is the module in the resource manager product that subscribes to all the events and it presents them in a consolidated manner to other event analyzer modules. The configuration database persistently stores the storage infra-structure configuration data, performance data, historical trends and also event data. The configuration database purges historical and event data based on user specified deletion policies.
- **SWEEPER Event Analyzer and Checkpoint Record Generator:** This module looks at the incoming stream of event data and filters out irrelevant events. For the relevant events, it determines the probability that the corruption may be correlated with this event and the scope (affected components) of the event. After the event analysis, this module generates a checkpoint record. The SWEEPER checkpoint record store is structured into two tiers of checkpoints. All checkpoint records generated by *Event Analyzer* are included in the lower tier and the records that have a high correlation probability with any type of corruption are promoted to the upper tier. The tier-ing notion can be extended from 2
- **SWEEPER Knowledgebase:** The knowledgebase consists of records with the following fields: a) List of event identifiers b) type or reason for data corruption c) probability of seeing an event (or a set of correlated events) in the case of data corruption. Corruption probability is represented as low, medium or high values because, in many cases, it is difficult for experts to specify exact probability values.
- **Problem Determination Scanners:** Failure detection can be done either manually or using an automated checking tool, *SWEEPER* leverages existing failure detection systems towards this purpose. Failure detectors also help in determining the type of failure (e.g., virus corruption) without pinpointing the system events that caused it.
- **SWEEPER Checkpoint Record Locator:** The *Checkpoint Record Locator* module orchestrates the overall recovery point identification process using one of the algorithms implemented by the *CDP Record Scanner*. The first task of the *Checkpoint Record Locator* module is to identify a subset of the checkpoint records that pertain to the current data corruption it is investigating. Towards this purpose, it queries the *Problem Determination Scanner* for the type and scope of the current corruption, and identifies the components that may be the cause of corruption. It uses the information to query the checkpoint database (indexed by type and scope) for checkpoint records that relate to the type of corruption and pertain to the affected components and creates a *Checkpoint Record Cache* for use by the *CDP Record Scanner*. The *Checkpoint Record Locator* allows the user to specify Recovery Time Objective or RTO (how long the user is willing to wait for the recovery process to complete) and Recovery Point Objective or RPO (how stale the data can be, and this is measure as a unit of time) and uses them along with the properties of the *Checkpoint Cache* to select one of the scanning algorithms implemented by the *CDP Record Scanner*. It queries the *CDP Record Scanner* with the *Cache* and scanning algorithm as input and receives a timestamp as the answer. It then uses the *CDP Recovery Module* and the *Problem Determination Scanner* to create and test the data according to the timestamp and returns the answer (corrupt/clean) of the *Problem Determination Scanner* to the CDP Record Scanner and receives a new timestamp. In each iteration of the procedure, it computes the total time taken by

the *Recovery Flow* and if it exceeds the RTO objective, it terminates with the most recent clean copy of the data as the recovered data. Also, for each timestamp that is checked, it computes the RPO (distance between the most recent clean copy and the most stale corrupt copy) and terminates if the RPO objective is met with the most recent clean copy of the data.

- **SWEEPER CDP Record Scanner:** The *CDP Record Scanner* implements an interface that takes as input a scanning strategy, a Checkpoint record, cache, a cursor (timestamp), and the integrity (corrupt or clean) of the cursor and returns a new cursor (timestamp) to check for data integrity. The separation of the recovery point identification algorithms from the checkpoint database allows *SWEEPER* to be flexible enough to include new search strategies in future. Also, since the search strategies are independent of *SWEEPER*, they can be implemented in other recovery point identification architectures as well.

2.4 Overall System Flow

We now describe the *Checkpoint generation* flow and *Fault-isolation or Recovery* flow that capture the essence of the *SWEEPER* architecture. The *Checkpoint Generation* flow captures the generation of checkpoint records during normal CDP system operation. The recovery process is initiated by system administrators when they determine that data corruption has occurred and this is captured by the *Fault-Isolation* flow.

Checkpoint Generation Flow:

- The CDP system generates CDP records (either logs or data copies) based on user defined policies.
- The Storage Resource Manager product monitors and aggregates various types of system events (hardware and software failure triggers, user action triggers etc).
- In parallel with the CDP record generation process, the *SWEEPER* event analyzer module analyzes the system events and generates a checkpoint record for each relevant event. The checkpoint records are logically co-related with the CDP records via timestamps.
- The checkpoint record generator a) leverages the information in the expert knowledge base b) correlates the information in the event stream and c) traverses the configuration resource graph, to index the checkpoint record with the scope of the event and its correlation probability with each type of corruption.

Fault-Isolation Flow:

- The Problem Determination Scanner or a human detects that data corruption has occurred and identifies its scope and type (e.g., virus, hardware).
- A query is posed to the *SWEEPER Checkpoint Record Locator* module by the system administrator with RTO and RPO as input. The reason or type of the data corruption and its scope is also an input to the *Checkpoint Record Locator*. The input information is used by the *Checkpoint Record Locator* to create a list of checkpoint records that should be examined, and the scanning algorithm to be employed.
- The *SWEEPER CDP Record Scanner* is invoked with the list of checkpoint records and the scanning strategy as input. It determines the checkpoint record that should be examined next and returns the CDP record corresponding to it to the user.
- The user retrieves the CDP record from the CDP system and then either runs the appropriate diagnostics or manually examines the checkpoint record to see whether it is corrupted. This process is repeated by the user until a clean data copy is retrieved.

3 SWEEPER Checkpoint Log Processing Algorithms

The checkpoint log processing algorithms are implemented in the *Checkpoint Record Locator* and the *CDP Record Scanner* modules. The *Checkpoint Record Locator* iteratively queries the *CDP Record Scanner* for the next checkpoint that it should verify for correctness, whereas the *CDP Record Scanner* has the core intelligence for identifying the next eligible checkpoint. The iterative flow of *Checkpoint Record Locator* is presented in Fig. 3.

```
function locateCheckPoints
  start = 0, end = currTime
  while (timeSpent < RTO) AND (RPO not met)
    currCopy = scanRecordsAlgoType(start, end)
    if currCopy is clean
      start = timestamp of currCopy
    else
      end = timestamp of currCopy
    end while
  end locateCheckPoints
```

Figure 3: Checkpoint Record Locator Flow

3.1 CDP Record Scanning Algorithms

We now present the scanning algorithms that contain the core intelligence of *SWEEPER*. As in Sec. 2.1, *N*

denotes the number of CDP logs, C_t is the cost of testing a data image for corruption, C_p is the cost of getting a PIT copy online, C_l is the cost of applying one CDP log on a data image and T_p is the number of CDP logs after which a PIT image is taken. Further, we use N_c to denote the number of checkpoint records in the relevant history.

3.1.1 Sequential Checkpoint Strategy

The *Sequential Checkpoint Strategy* starts from the first clean copy of data (Fig. 4) and applies the CDP logs in a sequential manner. However, it creates data images only for timestamps corresponding to some checkpoint record. The implementation of the *scanRecords* algorithm returns the first checkpoint record after the given start time. Hence, the number of integrity tests that *Checkpoint Record Locator* needs to perform using the *scanRecordsSequential* method is proportional to the number of checkpoint records N_c and not on the number of CDP logs N . The worst-case cost of creating the most current clean data image by the *Sequential Scanning Strategy* is $N_c C_t + C_p + N C_L$.

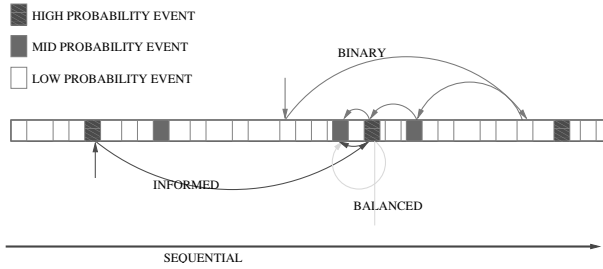


Figure 4: Search Strategies: Sequential follows a straight path. Binary Search reduces space by half in each step, Informed follows a strict order between probabilities, Balanced behaves in a probability-weighted Binary Search fashion.

3.1.2 Binary Search Strategy

We use the observation that corruption errors are not transient in nature and hence, if data is found to be corrupt at any time T_i , the data would remain corrupt for all timestamp $T_j > T_i$. This order preservation in corruption testing allows us to partition the search space quickly. We use the intuition of binary-search algorithms that partitioning a search space into two equal sized partitions leads one to converge to the required point in logarithmic time steps instead of linear number of steps. Hence, for a search space with $N_c(t)$ checkpoints at any given time t , *scanRecordsBinary* returns the timestamp corresponding to the $(N_c(t)/2)^{th}$ checkpoint record for inspection. If the data corresponding to the timestamp is corrupt, we recursively search for corruption in the timestamps between the 0^{th} and $(N_c(t)/2)^{th}$

checkpoints. On the other hand, if the data is clean, our inspection window is now the timestamps between the $(N_c(t)/2)^{th}$ and the $N_c(t)^{th}$ checkpoint records. It is easy to see that since the inspection window reduces by a factor of 2 after every check, we would complete the search in $\log N_c$ steps and the total time (expected as well as worst case) spent in *recovery point identification* is given by $\log N_c(C_t + C_p + C_l T_p/2)$.

3.1.3 Informed Search Strategy

While identifying the next timestamp to test for corruption, the *Binary Search* strategy selects the next checkpoint without taking into account the probability that the particular checkpoint was correlated with corruption. Our next strategy, called the *Informed* strategy, uses the probabilities associated with the checkpoint records to decide the next timestamp to examine. At any given time, it figures out the checkpoint j that has the highest likelihood $(p_{i,j})$ of being correlated with the corruption c_i . Hence, for cases where data corruption is associated with a rare event, the search may terminate in a constant number of steps (constant number of timestamps are examined) or take upto N_c steps in the adversarial worst case. However, as long as the probability $p_{i,j}$ of a particular checkpoint being the cause of corruption is uncorrelated with the time of its occurrence, the search would still reduce the space exponentially and is expected to terminate in logarithmic steps. Formally, we have the following result for the expected running time of *Informed Search*.

Theorem 1 *The Informed Search Strategy identifies the most recent uncorrupted data in $O(\log N_c(C_t + C_p + C_L T_p/2))$.*

Proof: To prove the result, it is sufficient to prove that the search is expected to examine only $O(\log N_c)$ timestamps before it finds the most recent uncorrupted entry. It is easy to see that the average cost of testing a given timestamp is given by $C_t + C_p + C_L T_p/2$.

Observe that as the highest probability checkpoint is uncorrelated with its timestamp, each of the N_c checkpoint records are equally likely to be examined next. Further, if the i^{th} checkpoint is examined, it divides the search space into two partitions, and we need to examine only one of them after the check. Hence, the recurrence relation for the search algorithm is given by

$$T(N_c) \leq \frac{1}{N_c}(T(N_c - 1) + T(1) + \dots + T(i) + T(N_c - i) + \dots + T(1) + T(N_c - 1)) \quad (3)$$

One may observe that $T(N_c) = \log N_c$ satisfies the recurrence relation. To verify, note that the right hand side of the equation reduces to $1/N_c \log N_c!$, if $T(N_c)$ is replaced by $\log N_c$. Hence, we only need to show that $T(N_c) < c \log N_c$ for some constant c

i.e., $\frac{\log(N_c!)^2}{N_c} < c \log N_c$
i.e., $\log(N_c!)^2 < c N_c \log N_c$
i.e., $\log(N_c!)^2 < c \log N_c^{N_c}$
which holds for $c = 2$ by using the fact that $N_c!^2 > N_c^{N_c}$. This completes the proof. ■

3.1.4 Balanced Search Strategy

The *Informed Search* strategy attempts to find the corruption point quickly by giving greater weightage to checkpoints that have a higher correlation probability with the corruption. On the other hand, *Binary Search* strategy prunes the space quickly oblivious to the probabilities associated with the checkpoint records. We combine the key idea of both these heuristics to design the *scanRecordBalanced* algorithm (Fig. 5). The algorithm computes the total search time in an inductive manner and selects the checkpoint record that minimizes the expected running time.

```

algorithm scanRecordsBalanced
  P(L) = 0; P(R) = 1; currMin = ∞
  for i in start to end
    T(i) = P(L) log(i-start) + Pi,j + P(R) log(end-i)
    P(L) = P(L) + Pi,j; P(R) = P(R) - Pi,j
    if T(i) < currMin
      currMin = T(i)
  end for
  return checkpoint of currMin
end scanRecordsBalanced

```

Figure 5: Balanced Scanning Algorithm

Intuitively speaking, the *Balanced* strategy picks the checkpoint records that (a) are likely to have caused the corruption and (b) partition the space into two roughly equal-sized partitions. The algorithm strikes the right balanced between partitioning the space quickly and selecting checkpoints that are correlated with the current corruption c_j . We have the following optimality result for the balanced strategy.

Theorem 2 *The balanced strategy minimizes the total expected search time required for recovery.*

Proof: In order to prove the result, we formulate precisely the expected running time of a strategy that picks a checkpoint record i for failure j . The expected running time of the strategy is then given in terms of the size and probabilities associated with the two partitions L and R .

$$T(N) = P(L)T(L) * C_{tot} + P_{i,j} * C_{tot} + P(R)T(R) * C_{tot} \quad (4)$$

where $P(L)$ and $P(R)$ are the accumulated probabilities of the left and right partitions respectively, and C_{tot} is

the total cost of creating and testing data for any given timestamp. Using the fact that $T(L)$ can be as high as $O(\log |L|)$ in the case where all checkpoint records have equal probabilities, we modify Eqn. 4 by

$$T(N) = P(L) \log |L| * C_{tot} + P_{i,j} * C_{tot} + P(R) \log |R| * C_{tot} \quad (5)$$

Hence, the optimal strategy minimizes the term on the right hand side, which is precisely what our balanced strategy does (after taking the common term C_{tot} out from the right hand side). This completes the proof. ■

4 Implementation

We have implemented *SWEEPER* as a pluggable module in a popular SRM (Storage Resource Manager) product, and it can use any CDP product. We use the SRM's *Event Bus* to drive the checkpoint record generation process in *SWEEPER*. The output of *SWEEPER* is a timestamp T_e that is fed to the *CDP Recovery Module*, and the *CDP System* rolls back all updates till time T_e . Since we can leverage many components from the SRM, the only components we needed to implement for *SWEEPER* were *SWEEPER Event Analyzer and Checkpoint Generator*, *SWEEPER Knowledge Database*, *SWEEPER CDP Record Scanner*, and *SWEEPER Checkpoint Record Locator*. Details of the algorithms implemented by *SWEEPER CDP Record Scanner* and *SWEEPER Checkpoint Record Locator* have been presented in Sec. 3 and we restrict ourselves to describing the implementation of the remaining components in this section. We start with the checkpoint record structure and its implementation.

4.1 Checkpoint Record Structure and Database

In our implementation, we use a relational database to store the checkpoint records for each event or event-set. Each checkpoint record consists of (i) *Checkpoint Record ID* (ii) *Timestamp* (iii) *Scope* (iv) *Failure Type*, and (v) *Correlation Probability*. *Checkpoint Record ID* is an auto-generated primary key and *Timestamp* corresponds to the time the event-set occurred (consists of year, month, day, hour, minute, second), and is synchronized with the CDP system clock. *Scope* describes whether the event-set is relevant for a particular physical device (e.g, switch, host, storage controller), or a software component (e.g., file system, database system), or a logical construct (e.g., file, table, zone, directory) and is used for quickly scoping the checkpoint records during recovery. *Failure Type* specifies the types of failure the checkpoint is relevant for is also used to scope the checkpoint records during recovery processing. The failure types (hardware failure, configuration error etc) are

listed in Table 2. *Correlation Probability* is computed as the probability that data corruption happened at the same time as the occurrence of the event or a set of events.

4.2 SWEEPER Knowledge Database

Event (e)	Source	$E(e c)$
Zoning Changes	Event Bus	Medium
LUN Masking	Event Bus	Medium
Access control changes	Event Bus	High
De-activating service	Event Bus	Medium
OS Update	Event Bus	Medium
Application Update	Event Bus	Medium
Driver Update	Event Bus	Medium
Firmware Update	Event Bus	Medium
New app deployment	Event Bus	Medium

(a)

Event (e)	Source	$E(e c)$
Writes in System Directory	DIR	High
Update of Registry startup entries	regedit	High
Writes to Files of specific type	DIR	Medium
Abnormal Network Activity	Perfmon, Nmap	Medium
Terminating Anti-virus Services	Win Event Viewer	Medium
High CPU Activity	Perfmon	Medium
SAN Activity	Event Bus	Low
Writes in registry and sysdir	DIR, regedit	High

(b)

Event (e)	Source	$E(e c)$
RSE Threshold Violation	SMART	High
SKE Threshold Violation	SMART	High
Mechanical Shock	SMART	Low
Opening of Unit	Manual	Low
Temperature Increase	SMART	Medium
Disk Scrubbers	Event Bus	Medium

(c)

Event (e)	Source	$E(e c)$
fsck	SRM	High
Exchange edbutil	Event Bus	High
Exchange eseutil	Event Bus	High
App Specific file updates	Event Bus	High
App Specific registry updates	Event Bus	High

(d)

Table 2: Monitored Events e_i of failure types (c_j): (a) Misconfiguration (b) Virus (c) Hardware related and (d) Application, along with their monitoring source and expert information ($P(e_i|c_j)$)

The *SWEEPER* checkpoint record generation centres around an expert information repository that we call the *SWEEPER Knowledge Database*. The *Knowledge Database* is structured as a table and a row in the table represents an event or a set of events e_i , and details the source of the event, the failure type(s) c_j relevant to the event and the correlation probability $E(e_i|c_j)$ of the event with each relevant failure type. A split-view of the *Knowledge Base* is presented in Table 1. We use expert information derived from literature to generate the *Knowledge Database*. We obtained common configuration related probabilities from the proprietary level two field support team problem database of a leading storage

company for storage area networks. We obtained hardware failure information from proprietary storage controller failure database of the same company. We obtained the virus failure information by looking at virus behavior for many common viruses at the virus encyclopedia site [28]. To elucidate with an example, for the corruption type of virus, we looked at the signature of the last 300 discovered viruses [28] and noted common and rare events associated with them. Based on how commonly an event is associated with a virus, we classified that event as having a low, medium or high probability to be seen in case of a virus attack. Because of the inherent noise in this expert data, we classify $P(e_i|c_j)$ only into low, medium and high probability buckets which correspond to probabilities of 0.1, 0.5 and 0.9 respectively.

The events listed in the *Knowledge Base* and monitored by the *Event Analyzer* can be classified into *Configuration Changes*: addition, update (upgrade/downgrade firmware, driver or software level) or removal of hardware and software resources and changes in security access control. These events are checkpointed as it is common for data corruption problems to occur when one upgrades a software level, or when one introduces a new piece of software or hardware resource. *Background Checking Processes*: successful or unsuccessful completion of background checking processes like virus scan, hardware diagnostics, filesystem consistency like fsck or application provided consistency checkers. These checkpoints provide markers for consistency in specific filesystems, databases or volumes. *Application Specific Changes*: Applications can provide hints about abnormal behaviour that may indicate corruption and *SWEEPER* allows one to monitor such events. *Hardware Failures*: checksum/CRC type errors, self diagnostic checks like SMART, warnings generated by SMART etc. *Performance Threshold Exceeding*: High port activity, high CPU utilization etc. Performance Threshold Exceeding events like high port utilization or high CPU utilization is usually a symptom of either a virus attack, or an application that has gone astray. *Meta-data Update Changes*: Changes in system directory etc. Abnormal *Meta-data updates* indicate application misbehavior, which, in turn, can potentially lead to data overwrite or corruption problems.

4.3 Event Filtering and Checkpoint Record Creation

The *SRM* has client agents that subscribe to various types of events from hosts, switches, storage controllers, file systems and applications. These events are consolidated and presented as part of an event bus. *SWEEPER* is only concerned about a subset of the events in the event bus and gets the list of relevant events from the *Knowledge Database* (Table 2). Once the rel-

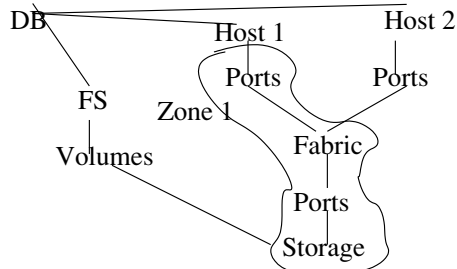


Figure 6: Resource Graph

event events are filtered, the *Event Analyzer* generates a checkpoint record for the event along with its timestamp. Tight synchronization between the *SWEEPER* event analyzer clock and the CDP system clock is not mandatory as the RPO granularity is no finer than 1 second. Hence, we perform hourly synchronization of the event analyzer clock with the CDP clock to ensure that the timestamps in the checkpoint records are reasonably accurate.

4.4 Checkpoint Record Scope Determination

The checkpoint record scope is useful in quickly filtering irrelevant checkpoint records, and thus, converge on the relevant CDP records. After the initial event filtering based on the information in the knowledgebase the checkpoint record generator examines the event type, and determines what data (files, DB tables or volumes) can be potentially affected by the event. We can only determine the scope for file/DB meta-data changes or configuration change related events. For other types of events the notion of scope is irrelevant. The value of scope is determined by traversing a resource graph. The resource graph information is stored in the systems resource manager configuration database. Fig. 6 shows a portion of the resource graph that is stored in the database. The nodes in the graph correspond to hardware and software resources, and the edges correspond to physical/logical connectivity or containment relationships. The basic structure of the resource graph is the SNIA SMI-S model. However, we have made extensions to this model to facilitate the modeling of application and database relationships. Fig. 6 illustrates how we traverse the resource graph when a configuration changing event occurs. If the user has added a new host and put its ports in Zone 1, then we determine all the storage ports (and the corresponding storage controllers) that are in zone 1. We then determine the storage volumes that are in those storage controllers and store the ids of the storage volumes in the scope field of the zoning event. During recovery, once the user has either

manually or via an automated tool identified a corrupt file/table/volume, we search the checkpoint records that list the file/table/volume in its scope.

4.5 Checkpoint Record Probability Determination

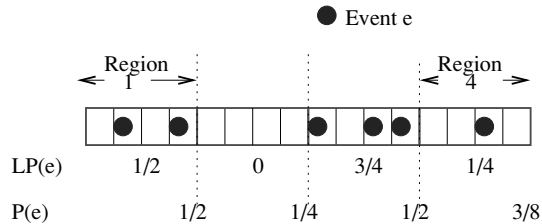


Figure 7: Estimating $P(e)$ with window size (W) of 4 using Exponential-weighted Averaging

The key feature of the *SWEEPER* architecture is associating correlation probabilities with events to help any search strategy in speeding up the recovery flow. For every event e_i and corruption c_j , the correlation probability $P(c_j|e_i)$ denotes the probability that the corruption c_j happened in a given time interval t given that e_i has happened at t . The correlation probability is estimated using the Bayesian.

$$P(c_j|e_i) = (P(e_i|c_j) \cdot P(c_j)) / P(e_i) \quad (6)$$

In order to compute the correlation probability, one needs to estimate $P(c_j)$, $P(e_i)$ and $P(e_i|c_j)$. $P(e_i)$ is estimated by looking at the event stream for the event e_i and using an exponential-weighted averaging to compute the average frequency of the event e_i . To elaborate further, the time line is divided into windows of size W each (Fig. 7), and for any window w_k , the number of occurrences n_i^k of each event e_i is noted. The local probability of event e_i in the window w_k (denoted by $LP(e_i^k)$) is calculated as n_i^k / W . In order to take into account the historical frequency of the event e_i and have a more stable estimate, the probability of an event e_i in the time window r_{k+1} ($k \geq 0$) is damped using the exponential decay function (Eq. 7). For the first window, the probability is same as the local probability.

$$P(e_i^{k+1}) = 1/2(P(e_i^k) + LP(e_i^{k+1})) \quad (7)$$

The $P(e_i|c_j)$ estimates are obtained from the *Knowledge Database*. The final parameter in Eqn. 6 is $P(c_j)$. However, note that whenever checkpoints are being examined to figure out the source of a corruption c_j , all the checkpoints would have the same common term $P(c_j)$. Hence, $P(c_j|e_i)$ is computed by ignoring the common term $P(c_j)$ for all the N events, and then normalizing the correlation probabilities so that $\sum_{i=1}^N P(c_j|e_i) = 1$. The correlation probabilities thus computed are stored in the checkpoint records for use in the recovery flow.

5 Evaluation

We conducted a large number of experiments to analyze the performance of our search algorithms and study the salient features of our checkpoint-based *SWEeper* architecture. We now report some of the key findings of our study.

5.1 Experimental Setup

Our experimental setup is based on the implementation described in Sec. 4 and consists of the architectural components in Fig 2. Since real data corruption problems are relatively infrequent events, we simulate a *Fault Injector* component in the interest of time. The *Fault Injector* takes as input a probabilistic model of faults and its possible signatures and generates one non-transient fault along with the signature. Since our *SWEeper* implementation is not integrated with any CDP system, we have also simulated a *CDP System Modeler and Problem Determination Scanner*. We have assumed that the *Problem Determination Scanner* can accurately identify if a copy of data is corrupt or not. The *CDP System Modeler* models the underlying storage and CDP system and provides the cost and time estimates for various storage activities like the time of applying a CDP log, making a PIT copy available for use or the time to test a snapshot for corruption. Finally, we use an *Event Generator* that mimics system activity. It takes as input a set of pre-specified events and their distribution and generates events that are monitored by the *SRM* and fed to the *SWEeper Event Analyzer* via the *SRM Event Bus*.

The key contribution of the checkpoint-based architecture is to correlate checkpoint records with corruption failures using the correlation probabilities $p_{i,j}$. Hence, we test our search algorithms for various distributions of $p_{i,j}$, which are listed below. We use synthetic correlation probability distributions because of the lack of authoritative traces of system and application events that would be applicable on a wide variety of systems. We have carefully inspected the nature of event distributions from many sources and use the following distributions as representative distributions of event-corruption probability.

- **Uniform Distribution:** This captures the setting where correlation information between checkpoints and corruption is not known and all system events are considered to be equally indicator of corruption.
- **Zipf Distribution:** The zipf distribution is commonly found in many real-life settings and capture the adage that the probability distribution is skewed towards a handful of events. For our experimental setting, this captures the scenario where only a few events are the likely causes of most of the failures.

- **2-level Uniform Distribution:** A 2-level (or generally speaking a k -level) uniform distribution has 2 (or k) types of event-types where all the events belonging to any given type have the same probability of having caused the corruption. However, certain event-types are more likely to have caused the error than other event-types and this is captured by having more than 1 level of probabilities.

One may note that the uniform and zipf distributions capture the two extremes in terms of skewness, that the correlation distribution $p_{i,j}$ may exhibit in practice. Hence, a study with these two extreme distributions not only capture many real settings, but also indicate the performance of the algorithms with other distributions as well.

Our first set of experiments studied the scalability and effectiveness of the *Checkpoint Generation Flow*. In the second set of experiments, we evaluate the various search strategies in the *Recovery flow*. We conducted experiments for scalability (increase in number of checkpoint records N_c) and their ability to deal with recovery time constraint (D_{rec}). We also study the usefulness of a 2-tier checkpoint record structure and the robustness of the *SWEeper* framework with false negatives (probability that the error is not captured in the event stream) and noisy data (error in correlation probability estimation). Since the *CDP system* was simulated, we had to manually fix the various parameters of the *CDP system* to realistic values. We kept the time taken to check the data corresponding to any given timestamp (C_t) as 10000 seconds, the time taken to get a PIT copy online (C_p) as 10 seconds, and the time taken to create the snapshot using the CDP logs ($C_l(T_i \% T_p)$) as 100 seconds.

5.2 Experimental Results

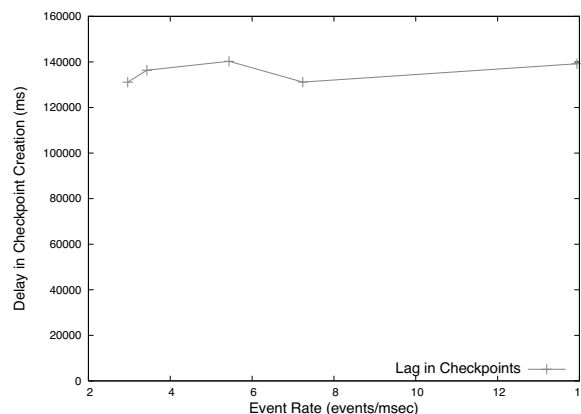


Figure 8: Lag in Checkpoint Records with Increasing Event Rate

We first investigated the scalability of our *Checkpoint Record Generation Flow* implementation to keep pace

with events as the SAN size grows. Hence, the *Event Generator* increases the number of resources managed by the Storage Resource Manager and generates more events, that are fed to the *Event Analyzer*. We observed that our *Event Analyzer* is able to efficiently deal with increased number of events (Fig. 8) and the lag in checkpoint record is almost independent of the event rate. We also observe that the checkpoint records lag by only 2mins and hence only the last 2 minutes of events may be unavailable for recovery flow, which is insignificant compared to the typical CDP windows of weeks or months that the recovery flow has to look into. The efficiency of the checkpoint flow is because (a) the computations in *Event Analyzer* (e.g., exponential-decay averaging) are fairly light-weight and (b) depend only on finite-sized window (Sec. 4.5), thus scaling well with number of events.

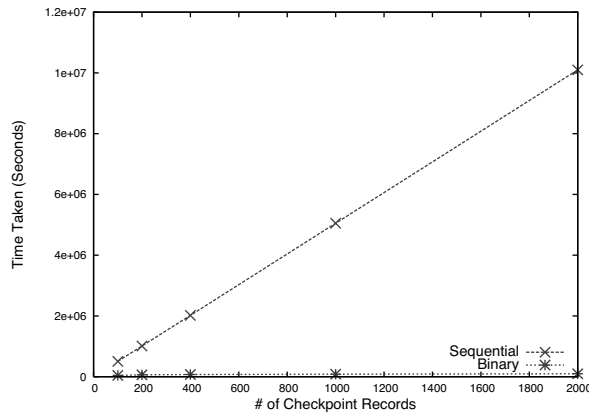


Figure 9: Recovery Time for Sequential and Non-sequential strategies under Uniform probability checkpoints

We next focus on the *Recovery Flow* and investigate the impact of using a non-sequential strategy as opposed to sequential checking. Fig 9 compares the time taken to find the most recent uncorrupted version of data by the sequential and the binary search strategies, for a uniform probability distribution. Since the probability of all checkpoints are equal, *Informed* as well as *Balanced* strategy have similar performance to *Binary Search*. For the sake of visual clarity, we only plot the performance of *Binary Search* algorithm. It is clear that even for small values of N , sequential algorithm fares poorly because of the $N/\log N$ running time ratio, and underlines the need for non-sequential algorithms to speedup recovery. For the remainder of our experiments, we only focus on non-sequential strategies and study their performance.

5.2.1 Performance under different distributions

We next studied the relative performance and scalability of the proposed non-sequential strategies for vari-

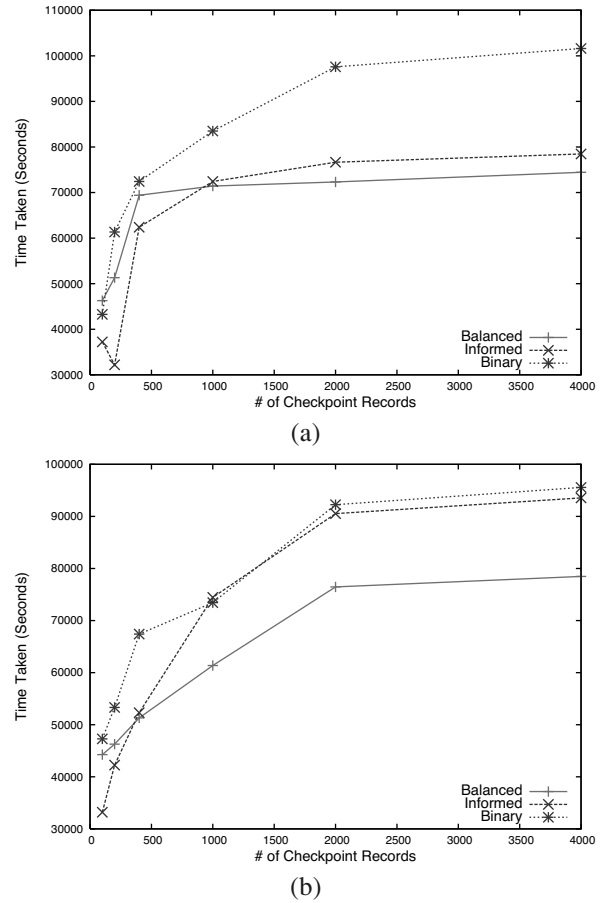


Figure 10: Recovery Time for Non-sequential strategies for (a) zipf distributed checkpoints and (b) 2-level uniform probability checkpoints

ous correlation probability distributions. Fig 10(a) and Fig 10(b) studies the performance of the three non-sequential strategies with increase in the number of checkpoint records under Zipf and 2-level uniform probability distribution for checkpoints. We observe that the *Balanced search strategy* always outperforms the *Binary search strategy*. This validates our intuition that since *Balanced* strategy partitions the search space by balancing the likelihood of corruption in the partitions rather than the number of checkpoint records it converges much faster than binary. The gap in performance is more for the Zipf distribution than 2-level Uniform distribution, as the more skewed Zipf distribution increases the likelihood of partitions with equal number of checkpoints to have very different accumulated correlation probabilities. The *Informed* search strategy performs well under Zipf distribution as it has to examine very few checkpoint records, before it identifies the recovery point. For small N , *Informed* even outperforms the *Balanced* strategy, which takes $O(\log N)$ time, while *Informed* runs in small number of constant steps, with very high proba-

bility. Thus, if the operator has high confidence in the events associated with different types of corruption, *Informed* may be the strategy of choice. One may observe for the 2-level uniform correlation probability case that, as the number of checkpoints increase, the performance of *Informed* degrades to that of *Binary* search. This is because the individual probability of each checkpoint record (even the checkpoints associated with the higher of the 2 levels) falls linearly with the number of points and hence, convergence in constant steps is no longer possible and *Informed* search converges in $\log N$ time, as predicted by Theorem 1. These experiments bring out the fact that the greedy *Informed* strategy is not good for large deployments or when the number of backup images are large.

5.2.2 Data Currency and Execution Time Tradeoff

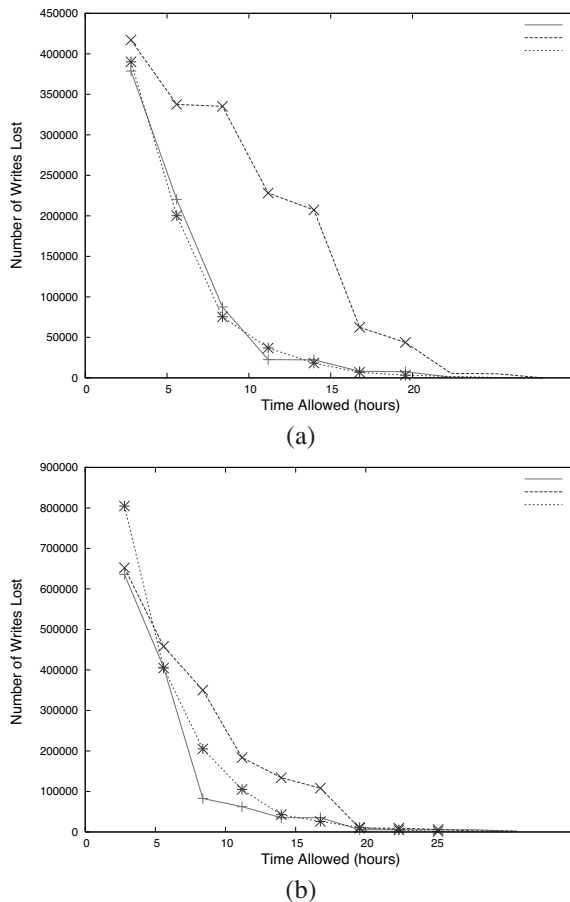


Figure 11: Data Currency for Non-sequential strategies with Recovery Time Constraint under (a) 2-level uniform distribution and (b) Zipf Distribution

We next modify the objective of the search algorithms. Instead of finding the most recent clean datapoint, the recovery algorithms now have a constraint on the recovery time and need to output a datapoint, at the end of the time

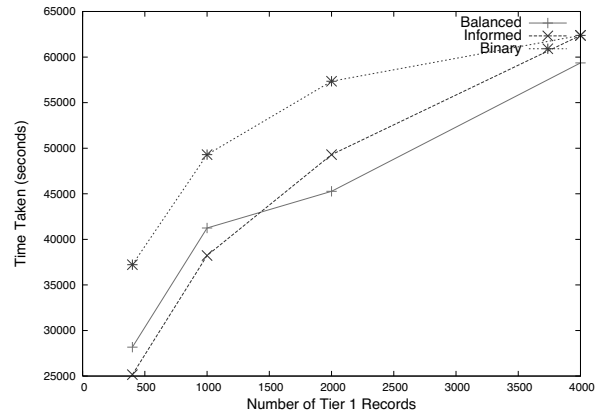


Figure 12: Recovery Time for Non-sequential strategies with 2-tier architecture

window. Fig 11(a) and Fig 11(b) examine the tradeoff between the recovery execution time and the data currency at the recovery point for 2-level uniform and Zipf distributions respectively. Data currency is measured in terms of lost write updates between the most recent uncorrupted copy and the copy returned by the algorithms.

The largest difference in performance under the two distribution is for the *Informed* strategy. It performs well under Zipf, since it starts its search by focusing on the few high probability checkpoints to quickly prune the search space. Under 2-level uniform distribution, it performs poorly as compared to the other strategies overall but more so when recovery time is less. This is because it evaluates the uniform probability events in a random order, which on an average leads to more unbalanced partitions, as compared to the other two strategies. Intuitively, both the *Binary* and *Balanced* strategies aim to reduce the unexplored space in each iteration. Hence, they minimize the distance of the error snapshot from the set of snapshots checked by them. On the other hand, *Informed* does not care about leaving a large space unexplored by it, and hence before it finds the actual error times, its best estimate of error time may be way off the actual error time. A similar observation can be made if the algorithms aim to achieve a certain (non-zero) data-currency in the minimum time possible. By reversing the axis of Fig. 11, one can observe that both *Binary* and *Balanced* strategies achieve significant reduction in data-currency fairly quickly, even though *Informed* catches up with them in the end. Hence, when the recovery time window is small, the use of *Informed* strategy is not advisable.

5.2.3 Checkpoint Selection using 2-tiers

We next investigate the impact of using a 2-tiered checkpoint record structure on the performance of the algo-

rithms. In a 2-tiered record structure, the algorithms execute on only a subset of checkpoint records consisting of high probability checkpoint records. The idea is that the recovery point is identified approximately by running the algorithms on the smaller set of records and then locate the exact recovery point using the checkpoints in the neighborhood. For the plot in Fig 12, checkpoints are assigned probabilities according to 2-level uniform distribution with 5% of checkpoints having 90% cumulative probability. Further, only 10% of total checkpoints are promoted to the high probability tier, and hence the skew in the high-tier checkpoints is much lower than a single tier checkpoint record structure. We observe in Fig. 12 that, as compared to a single tier checkpoint records (Fig 10(b)), the performance gap between binary and balanced reduces significantly. This is because the binary strategy reaps the benefit of operating only on the high probability checkpoint subset, making it closer in spirit to the balanced strategy. Thus, a 2-tiered checkpoint architecture, makes the probability oblivious *Binary Search* algorithm also probability-aware, by forcing it to operate only on checkpoints that have a high correlation probability with the corruption.

5.2.4 Robustness of the Algorithms with incomplete information

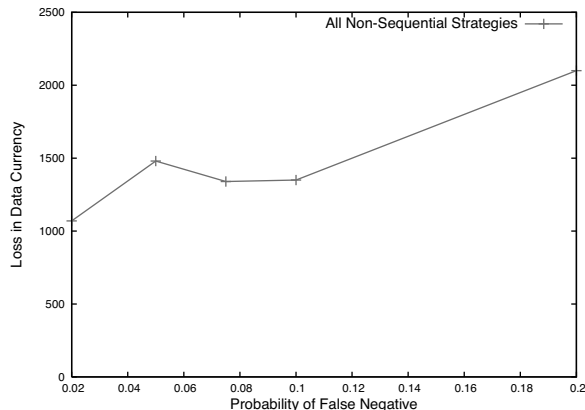


Figure 13: Data Currency Loss for Non-sequential strategies with Probability of False Negatives

We next investigate the robustness of our proposed algorithms to deal with silent errors; i.e. an error that does not generate any events. We model silent errors using false negatives (probability that the error is not captured in the checkpoint records). Fig. 13 studies the loss in data currency with increase in false negatives. For this experiment, the average number of CDP logs between any two events was kept at 2000, and we find that the non-sequential strategies are able to keep the data currency loss below or near this number even for a false negative of 20%. Our results show the correlation probability-

aware algorithms like *Balanced* and *Informed* are no worse than correlation probability-unaware algorithms like *Sequential* and *Binary*. Hence, even though these strategies factor the correlation probability while searching, they do not face significant performance penalty, when the error event is not captured in the checkpoints.

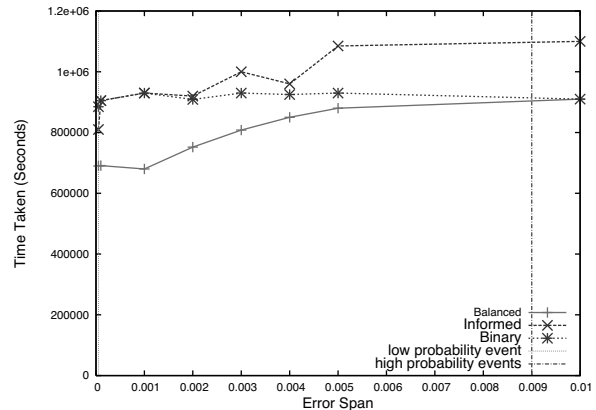


Figure 14: Recovery Time with Increasing Error for Non-sequential strategies

In practice, the expert-given values for $P(e_j|c_i)$ as well as the computed $P(e_j)$ have estimation errors and the algorithms should be robust enough to deal with noise in correlation probability estimates. Fig. 14 shows the recovery time taken by the various algorithms for the 2-level probability distribution, as the expert data and event stream becomes noisy. The noise is generated by adding a zero mean uniform distribution to the correlation probabilities, where the range of the noise is varied from 0 to the probability of the highest probability event. Note that the two vertical lines indicate the probability of the two types (levels) of events in the 2-level distribution and as the noise (error) approaches the right line, the standard deviation of the noise becomes greater than the mean of the original signal (or correlation probability). In such a situation, the correlation probabilities lose their significance as the complete data can be thought of as noise. The key observation in this study is the sensitivity of *Informed strategy* to the accuracy of correlation probabilities and the robustness of *Balanced Strategy* to noise. We observed that *Balanced Strategy* showed a graceful degradation with increase in error, degenerating to *Binary Strategy* even when the correlation probability had only noise, whereas *Informed Strategy* showed a steep increase in recovery time. This is not surprising, given that *Informed Strategy* only considers the individual correlation probability and suffers with increased estimation error. On the other hand, *Balanced Strategy* takes into account cumulative probabilities, which are not effected as much by the zero mean noise until noise dominates the original signal. Even in the case where error dominates

the original probabilities, *Balanced* performs as well as the *Binary* strategy underlining its usefulness as an effective, versatile and robust strategy.

6 Discussion and Related Work

We present in this paper a scalable architecture and efficient algorithms that help administrators deal with data corruption, by quickly rolling back to an earlier clean version of data. The related work in this area can be broadly classified into techniques for (a) *Data Resiliency and Recovery*, (b) *Event Monitoring and Logging*, (c) *Indexing*: Correlating events with corruption and using the correlation values as index and (d) *Searching* the event logs for quick identification of the failure point.

Continuous Data Protection (CDP) solutions deal with data corruption by allowing an administrator to roll back at a granularity that is much finer than what was possible with traditional continuous copy or backup solutions. CDP solutions have been proposed at file level [26] and at block level in the network layer [16, 21, 30, 15]. Versioning file systems [25, 20] that preserve earlier versions of files also provide the CDP functionality. CDP logs allow users to revert back to earlier data versions but leaves the onus of determining a recovery point on the administrator. A brute-force approach examining all CDP logs could lead to unacceptably high recovery time. To alleviate this, some products such as [16, 21, 30] incorporate *Event Monitoring and Logging* with the basic *Data Resiliency* mechanisms and allow applications to record specific checkpoint records in the CDP log, which then serve as landmarks in the log stream to narrow down the recovery point search. However, these solutions present no *Indexing and Searching* mechanism and the administrator has to devise a search strategy between the checkpoints, where all checkpoints are as equally likely to be associated with the failure. Our work builds on existing solutions by (i) using system events along with application events to generate checkpoints, (b) attaching correlation probabilities with the checkpoints, and (c) providing CDP log processing algorithms that use these probabilities intelligently to automatically identify a good recovery point quickly.

The use of *Searching* and Mining techniques on an *Event Log* has been very popular in the area of problem [9] detection and determination in large scale systems. Numerous problem analysis tools [29], [32], [13] have been proposed that aid in the process of automating *Searching* the logs for problem analysis. A major issue in application of these techniques in large systems is the complexity of the event collection and subsequent analysis. Xu et al. [31] propose a flexible and modular architecture that enables addition of new analysis engines with relative ease. Kiciman et al. [12] use anomalies

in component interactions in an Internet service environment for problem detection. Chen et al. [4] use a decision tree approach for failure diagnosis in eBay production environment. File system problem determination tools have been proposed [32] [13] that monitor system calls, file system operations and process operations to dynamically create resource graphs. Once the system administrator detects that there is a problem, these tools help to quickly identify the scope of the problem with respect to the affected files and processes. Similarly, Chronus [29] allows users to provide customized software probes that help in identifying faulty system states by executing the probe on past system states that have been persisted on disk. They select past system states (virtual machine snapshots for a single machine) using binary search. Hence, Chronus addresses two of the problems, namely *Event Monitoring* and *Searching* that *SWEEPER* handles. Even though it solves a completely different problem from *SWEEPER*, Chronus is most similar to *SWEEPER* in spirit. However, Chronus does not distinguish between different event checkpoints, whereas we index the checkpoints based on correlation probabilities and design a search strategy that uses this information for fast convergence. Further, we provide the user the flexibility to tradeoff on the data currentness/recovery time trade-off by appropriately setting the RTO and RPO values. Finally, the focus of our paper is on data corruption on disk in a distributed environment whereas Chronus deals with system configuration problems for a single machine.

The novelty of *SWEEPER* lies in integrating *Event monitoring*, *Checkpoint Indexing*, and new search techniques that are able to make use of the index (correlation) information. A possible weakness of *SWEEPER* lies in its inherent design that is based on checkpoint events: *SWEEPER* depends on events to provide hints for corruption and silent errors or noise in the event-corruption correlation may degrade *SWEEPER*'s performance. However, the proposed search algorithms do a balancing act between searching events with high correlation probabilities and pruning the search space in logarithmic time. Our study suggests the use of *Balanced Search* as the most robust strategy for efficient *Recovery Point Identification*. The *Balanced Search* strategy finds a good recovery time for silent errors by degenerating to *Binary Search*. The only scenario where *Balanced Search* is (marginally) outperformed is in the case of *probability skew*: a few checkpoints capture most of the correlation probability and *Informed Search* is the best performer. Our study makes a strong case of using search algorithms that use the correlation between system events and corruption to quickly recover from data corruption failures.

7 Conclusion

In this paper we presented an architecture and algorithms to quickly identify clean data in CDP record stream. The basic idea of our system is to monitor system events and generate checkpoint records that provide hints about potential data corruption. Upon discovering data corruption, system administrators can pass RTO and RPO requirements as input to our system, and it returns the most relevant checkpoint records. These checkpoint records point to locations in the CDP record stream that potentially will provide clean data. We use expert knowledge, resource dependency analysis, and event co-relation analysis to generate the checkpoint records. Upon the discovery of data corruption, system administrators can use *SWEEPER* to quickly identify relevant CDP records. We present different CDP record traversal algorithms to help traverse the CDP record stream to identify recovery points. Our studies suggest the use of *Balanced Search* strategy as the most robust strategy for efficient *Recovery Point Identification*. However, in an environment where checkpoints with high correlation probabilities are very few, the *Informed Search* strategy seems to be a good choice. Hence, the study emphasizes the need for either the log processing algorithms or the checkpoint architecture to be aware of correlation probabilities of various events. Furthermore, our study conclusively establishes the need for non-sequential search mechanisms to quickly identify good recovery points in a CDP environment as opposed to a linear sequential scanning of checkpoints.

References

- [1] D. Agrawal, J. Giles, K. Lee, K. Voruganti, K. Filali-Adib. Policy-Based Validation of SAN Configuration. In *IEEE Policy*, 2004.
- [2] M. Baker, M. Shah *et al.* A Fresh Look at the Reliability of Long-term Digital Storage. In *EuroSys*, 2006.
- [3] M. Banikazemi, D. Poff and B. Abali. Storage-Based Intrusion Detection for Storage Area Networks (SANs). In *IEEE/NASA MSST*, 2005.
- [4] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. A Statistical Learning Approach to Failure Diagnosis. In *ICAC*, New York, NY, May 2004.
- [5] DB2 Universal Database - High Availability Disaster Recovery (HADR). At www.ibm.com/software/data/db2/udb/hadr.html.
- [6] DMTF-CIM. At www.dmtf.org/standards/cim/
- [7] EMC SRDF Family. At <http://www.emc.com/products/networking/srdf.jsp>
- [8] M. D. Flouris and A. Bilas. Clotho: Transparent data versioning at the block I/O level. In *Proceedings of NASA/IEEE Conference on Mass Storage Systems and Technologies*, 2004.
- [9] J. L. Hafner, V. Deenadhayalan, K. K. Rao and J. A. Tomlin. Matrix Methods for Lost Data Reconstruction in Erasure Codes. In *USENIX Conference on File And Storage Technologies*, FAST 2005.
- [10] IBM General Parallel File System. At <http://www-03.ibm.com/systems/clusters/software/gpfs.html>.
- [11] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *n Third Usenix Conference on File and Storage Technologies (FAST2004)*, 2004.
- [12] E. Kiciman and A. Fox. Detecting Application-Level Failures in Component-based Internet Services. In *IEEE Transactions on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks (invited paper)*, Spring 2005.
- [13] S. King and P. Chen. Backtracking Intrusions. In *SOSP*, 2003.
- [14] A. Kochut and G. Kar. Managing Virtual Storage Systems: An Approach Using Dependency Analysis. In *IFIP/IEEE IM 2003*, 2003.
- [15] G. Laden, P. Ta-Shma, E. Yaffe, M. Factor and S. Fienblit. Architectures for Controller Based CDP. In *Usenix FAST*, 2007.
- [16] Mendocino Software. <http://www.mendocinosoft.com>
- [17] Online survey results: 2001 cost of downtime. Eagle Rock Alliance Ltd, Aug. 2001. At <http://contingencyplanningresearch.com/2001Survey.pdf>
- [18] Open Source Rule Engines in Java. At <http://java-source.net/open-source/rule-engines>.
- [19] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based Intrusion Detection: Watching Storage Activity For Suspicious Behavior. In *USENIX Security Symposium*, 2003.
- [20] K. M. Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *USENIX Conf. on File And Storage Technologies*, FAST 2004.
- [21] <http://www.revivio.com>.
- [22] SAP R/3 FAQ - ABAP/4 Dictionary. www.sappoint.com/faq/faqabdic.pdf.
- [23] Self-Monitoring, Analysis and Reporting Technology (SMART) disk drive monitoring tools. At <http://sourceforge.net/projects/smartmontools/>.
- [24] Storage Networking Industry Association. An overview of today's Continuous Data Protection (CDP) solutions. At http://www.snia.org/tech_activities/dmf/docs/CDP_Buyers_Guide_20050822.pdf.
- [25] C. Soules, G. Goodson, J. Strunk and G. Ganger. Metadata efficiency in versioning file systems. In *Second Usenix Conference on File and Storage Technologies, (FAST 2003)*, San Francisco, USA, 2003.
- [26] IBM Tivoli Continuous Data Protection for Files. www-306.ibm.com/software/tivoli/products/continuous-data-protection/
- [27] Veritas Volume Replicator. At www.symantec.com.
- [28] Virus Encyclopedia. At <http://www.viruslist.com/viruses/encyclopedia>
- [29] A. Whitaker, R. Cox and S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, 2004.
- [30] XOssoft Backup and Recovery Solution. At <http://www.xosoft.com>.
- [31] W. Xu, P. Bodik, D. Patterson. A Flexible Architecture for Statistical Learning and Data Mining from System Log Streams. In *IEEE ICDM'04*, Brighton, UK, November 2004.
- [32] N. Zhu and T. Chiueh. Design, Implementation and Evaluation of Repairable File Service. In *DSN*, 2003.

Notes

¹Work done while at IBM India Research.

Using Utility to Provision Storage Systems

John D. Strunk[†], Eno Thereska*, Christos Faloutsos[†], Gregory R. Ganger[†]

[†]*Carnegie Mellon University*

**Microsoft Research, Cambridge UK*

Abstract

Provisioning a storage system requires balancing the costs of the solution with the benefits that the solution will provide. Previous provisioning approaches have started with a fixed set of requirements and the goal of automatically finding minimum cost solutions to meet them. Such approaches neglect the cost-benefit analysis of the purchasing decision.

Purchasing a storage system involves an extensive set of trade-offs between metrics such as purchase cost, performance, reliability, availability, power, etc. Increases in one metric have consequences for others, and failing to account for these trade-offs can lead to a poor return on the storage investment. Using a collection of storage acquisition and provisioning scenarios, we show that utility functions enable this cost-benefit structure to be conveyed to an automated provisioning tool, enabling the tool to make appropriate trade-offs between different system metrics including performance, data protection, and purchase cost.

1 Introduction

Whether buying a new car or deciding what to eat for lunch, nearly every decision involves trade-offs. Purchasing and configuring a storage system is no different. IT departments want solutions that meet their storage needs in a cost-effective manner. Currently, system administrators must rely on human “expertise” regarding the type and quantity of hardware to purchase as well as how it should be configured. These recommendations often take the form of standard configurations or “rules of thumb” that can easily lead to an expensive, over-provisioned system or one that fails to meet the customer’s expectations. Because every organization and installation is unique, a configuration that works well for one customer may provide inadequate service or be too expensive for another. Proper storage solutions account not only for the needs of the customer’s applications but also the customer’s budget and cost structure.

Expressiveness →		
Mechanisms	Goals	Utility
RAID-5 64 kB stripe size	500 IO/s 5 “nines”	U(revenue, costs)
Manual configuration	Provisioning using fixed requirements	Provisioning using business objectives

Figure 1: Utility provides value beyond mechanism-based and goal-based specification – Moving from mechanism-based specification to goal-based specification allowed the creation of tools for provisioning storage systems to meet fixed requirements. Moving from goal-based to utility-based specification allows tools to design storage systems that balance their capabilities against the costs of providing the service. This allows the systems to better match the cost and benefit structure of an organization.

Previous approaches to provisioning [5–7] have worked to create minimum cost solutions that meet some predefined requirements. Purchasing the cheapest storage system that meets a set of fixed requirements neglects the potential trade-offs available to the customer, because it separates the analysis of the benefits from that of the costs. In this scenario, system administrators are forced to determine their storage requirements (based on their anticipated benefits) prior to the costs becoming apparent — those “5 nines” are not free. Our goal is to combine these two, currently separate, analyses to produce more cost-effective storage solutions.

While some requirements may be non-negotiable, most are flexible based on the costs required to implement them. Few organizations would say, “I need 5000 IO/s, and I don’t care what it costs.” Performance objectives are related to employees’ productivity and potential revenue. Data protection objectives are related to the cost, inconvenience, and publicity that come from downtime and repair. These underlying costs and benefits determine the ROI an organization can achieve, and a provisioning tool needs to consider both.

We propose using *utility functions*, instead of fixed requirements, as a way for the system administrator to

communicate these underlying costs and benefits to an automated provisioning tool. Figure 1 shows a spectrum of storage configuration methods from setting low-level mechanisms to communicating costs and benefits using utility. Utility functions provide a way to specify storage requirements in terms of the organization's cost/benefit structure, allowing an automated tool to provide the level of service that produces the most cost-effective storage solution for that environment. A utility function expresses the desirability of a given configuration — the benefits of the provided service, less the associated costs. The objective of the tool then becomes maximizing utility in the same way previous work has sought to minimize system cost.

This paper describes utility, utility functions, and the design of a utility-based provisioning tool, highlighting the main components that allow it to take high-level objectives from the administrator and produce cost-effective storage solutions. It describes the implementation and provides a brief evaluation of a prototype utility-based provisioning tool. The purpose of this tool is to show the feasibility of using utility to guide storage system provisioning. After this provisioning tool is shown to efficiently find good storage solutions, three hypothetical case studies are used to highlight several important benefits of using utility for storage provisioning. First, when both costs and benefits are considered, the optimal configuration may defy the traditional intuition about the relative importance of system metrics (e.g., performance vs. data protection). Second, utility can be used to provision systems in the presence of external constraints, such as a limited budget, making appropriate trade-offs to maximize benefits while limiting the system's purchase cost. The third case study provides an example of how changes in the cost of storage hardware can affect the optimal storage design by changing the cost/benefit trade-off — something not handled by minimum cost provisioning.

2 Background

A goal of many IT departments is to support and add value to the main activities of an organization in a cost-effective manner. This endeavor involves balancing the quality and benefits of services against the costs required to provide them. In provisioning a storage system, the administrator attempts to balance numerous objectives, including *performance* (e.g., bandwidth and latency), *data protection* (e.g., reliability and availability), *resource consumption* (e.g., capacity utilization and power consumption), *configuration manageability* (e.g., configuration stability and design simplicity), and *system cost*. The current best practices for this problem are

based on simple “rules of thumb” that create classes of storage to implement different points in this rich design space [31]. For example, an administrator may define storage classes, such as “business critical” or “archival,” to denote particular data encodings (e.g., RAID level) and pools of disks to use for storing the data. The “business critical” class may require both high performance and a high level of reliability, while “archival” data could be placed on lower performance, lower cost, yet still reliable storage.

The task of the administrator is made more difficult with the emergence of cluster-based storage that provides the opportunity for a wider range of data placement and encoding options. For instance, both FAB [29] and Ursa Minor [1] allow the use of arbitrary m -of- n erasure codes for data protection.¹ The PASIS protocol [17] used by Ursa Minor provides the ability not only to use erasure coding to store data, but it allows the encoded data fragments to be stored onto an arbitrary subset, l , of the system's storage nodes. For example, with a “2-of-3 declustered across 4” encoding, a data object could be stored using a 2-of-3 scheme with the data spread across storage nodes one, four, five, and six.

It is difficult to determine which data encodings cost-effectively produce the proper mix of storage objectives for a particular class of data. A vendor's support organization may provide some guidance based on other installations, but the decision is ultimately up to the administrator. Unfortunately, the system administrator is ill-equipped to make this decision. He may understand that increasing the difference between m and n will provide higher reliability and potentially lower performance, but quantifying these metrics is difficult, even for experts in the field. Table 1 illustrates the general effects that changes to the data encoding parameters cause. A change to any encoding parameter affects nearly all of the system metrics — some for the better and some for the worse.

Gelb [16], Borowsky et al. [10], and Wilkes [37] have argued that administrators should be concerned with high-level system metrics, not the underlying mechanisms (e.g., the levels of performance and data protection, not the values of m , n , and l) and that a management system should automatically choose the mechanisms that produce the desired level of each metric. Unfortunately, the settings that maximize one metric are likely to severely impair others. For example, configurations that maximize reliability tend to consume a large amount of capacity, raising the hardware cost and sacrificing some performance. This lack of independence brings the need to arbitrate between, or trade off, one metric (e.g., power)

¹With m -of- n erasure codes, a data block is divided into n fragments, any m of which can be used to reconstruct the original data.

Metric	$m \uparrow$	$n \uparrow$	$l \uparrow$
Availability	\downarrow	\uparrow	\downarrow
Reliability	\downarrow	\uparrow	\downarrow
Capacity consumed	\downarrow	\uparrow	—
Read bandwidth	\downarrow	\uparrow	\uparrow
Read latency	\uparrow	—	\downarrow
Write bandwidth	\uparrow	\downarrow	\uparrow
Write latency	—	\uparrow	\downarrow

Table 1: General effects of encoding parameters on system metrics – This table shows the general effects on various system metrics caused by increasing the data encoding parameters, m , n , or l . The magnitude of these effects vary considerably and are difficult to quantify without detailed models. Making the proper encoding choice manually is difficult because changing a single parameter affects nearly all the metrics. A system that is able to choose these parameters automatically must be able to make trade-offs across metrics.

against another (e.g., request latency). We show that utility functions provide a good method for administrators to provide automation tools with the necessary information to make these choices.

2.1 Utility

Utility is a value that represents the desirability of a particular state or outcome. This concept is common in both economics (to explain consumer preferences) and in decision theory [20] (as a method for weighing alternatives). The main feature we use in this paper is its ability to collapse multiple objectives (e.g., performance and reliability) into a single axis that can be used to compare alternatives. When presented with a suitable utility function, an automated tool can use the utility values to compare storage designs in a manner consistent with the desires of the system administrator.

To use utility to guide storage provisioning, it is necessary to have a utility function that is able to evaluate a potential storage configuration and produce a single value (its utility) that can be compared against other candidate configurations. The optimal configuration is the one with the highest utility value. The utility value for a configuration should be influenced by the system metrics that are important to the administrator. For example, configurations with high performance would have higher utility values than those with low performance — likewise for availability and reliability.

Examining system metrics in isolation, one could use the actual metric as the utility value. For example, setting $Utility = Bandwidth$ would cause the provisioning system to prefer configurations with high bandwidth over those with low bandwidth. The goal of utility, how-

ever, is to combine all relevant system metrics into a single framework. The various metrics cannot simply be summed; they must be combined in a manner that captures their relative importance. This restriction requires the metrics to be normalized or scaled relative to each other. Experience suggests that the easiest method for normalizing these different metrics is via a common scale that has meaning for each metric. One such scale is money (e.g., dollars). Since each storage metric has an effect on the service provided, it impacts an organization's business, and this business impact can be expressed in dollars. For example, performance (throughput) affects the number of orders per second that an e-commerce site can handle, and loss of availability causes lost business and decreased productivity. By expressing each source of utility (e.g., performance, data protection, and system cost) in dollars, they can be easily combined.

System administrators can create utility functions by assessing the objectives of the storage system from a business perspective. This type of analysis tends to yield a set of functions related to different aspects of the storage service. For example, one function may express e-commerce revenue as a function of performance and availability. Another could describe the costs associated with a data-loss event. Other, more direct costs can be incorporated as well, such as the cost of power and cooling for the storage system or its purchase cost. These separate functions, expressed in the same “units” of currency can be summed to produce a single utility function for use in automated provisioning. While some may question the ability of the administrator to evaluate their storage needs in such business terms, we believe this approach is just providing a more formal framework for an analysis that is already performed. The system administrator is already required to justify purchase and configuration decisions to upper-level management — a conversation that is surely framed in a business context.

2.2 Related work

The problem of replica placement has been studied extensively in the context of the File Assignment Problem [12]. A similar problem has been examined for placing replicas in content distribution networks (e.g., Baev and Rajaraman [9], Tang and Xu [32]). While some of the work on FAP has examined detailed storage performance models [38], the content distribution work is largely concerned with communication costs and networks where constraints are linear, a condition that does not hold for either the performance models nor the utility functions. Additionally, because data encodings, in addition to locations, are selected as part of the optimization,

replica placement is just one part of the overall provisioning task.

Storage provisioning and configuration tools, including Minerva [2], Ergastulum [5], and the Disk Array Designer [7], have largely been targeted at creating minimum cost designs that satisfy some fixed level of performance and data protection. Our work builds on these tools by removing the fixed requirements, and, instead, using utility as the objective function. Using utility allows our system to make automatic trade-offs across the various storage metrics.

In the push toward automation, the notion of using utility to guide self-tuning and autonomic systems is becoming more popular. Kephart and Walsh [23] provide a comparison of event-condition-action (ECA) rules, goals, and utility functions for guiding autonomic systems. They note that both goals and utility are above the level of the individual system mechanisms and that utility provides a level of detail over goal-based specification that allows conflicting objectives to be reconciled automatically.

Utility has been applied to scheduling batch compute jobs to maximize usefulness for the end user in the face of deadlines [19] or where the results of many dependent jobs are needed [8]. For web-based workloads, Walsh et al. [35] describe a system that allocates server resources to control the response time between two different service classes.

There has also been work on designing cost-effective disaster recovery solutions, trading off solution costs with expected penalties for data loss and downtime [14, 21, 22]. This work has effectively used utility to trade off the costs of data protection mechanisms against the penalties when data is lost, creating minimum (overall) cost solutions for disaster recovery. This result lends support to the notion of using business costs as the basis for evaluating storage solutions.

3 Provisioning with utility

Provisioning a storage system to provide the most value for its owner requires the ability to make appropriate trade-offs among competing objectives. A provisioning tool begins with an initial *system description* that contains the parameters of the problem, including the available hardware types as well as the workload and dataset descriptions. The tool needs three main components to automatically create cost-effective storage systems using utility. The *system models* analyze a candidate configuration, annotating it with one or more metrics. The *utility function* uses these metrics to evaluate the configuration, assigning it a single utility value that indicates the de-

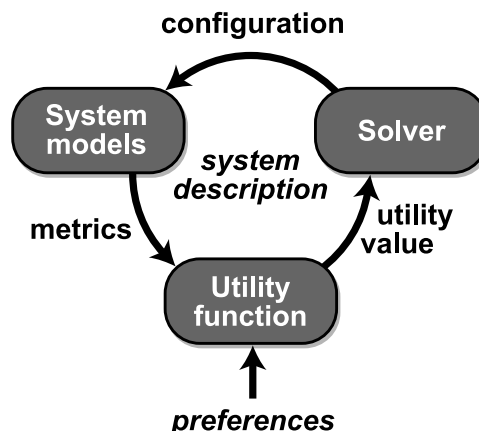


Figure 2: Overview of a utility-based provisioning tool – The solver produces candidate system configurations. The system models annotate the configurations with system, workload, and dataset metrics. The utility function uses the administrator’s preferences to rank the annotated configurations by assigning a utility value to each.

sirability of the configuration. The *solver* generates new candidate configurations based on the feedback provided by these utility values. Figure 2 shows the interaction between these three main components.

3.1 System description

The system description provides the baseline components, such as the clients, workloads, datasets, and storage nodes, that are available to the provisioning system. These components are the building-blocks that the provisioning tool uses when creating solutions. The workloads describe the demands placed on stored data by applications. Workloads are statically assigned to clients, and their I/O requests are directed at specific datasets. The main task of the provisioning tool is to assign datasets to storage nodes, choosing the data distribution that maximizes utility. A *candidate configuration* describes the mapping of each dataset onto the storage nodes. This configuration information, combined with the system description defines a provisioned storage system that can be evaluated by the system models.

3.2 System models

System models translate from the low-level mechanism-centric configuration into a description that contains high-level system metrics, such as the performance or data protection characteristics that the design is expected to produce. For example, the configuration presented to

the system models may indicate that a dataset is encoded as “2-of-3 spread across storage nodes one, four, six, and seven.” An availability model may translate this into a metric that states “the dataset has a fractional availability of 0.9997.” Numerous projects have produced storage system models for performance [4, 25, 33, 34], data protection [3, 11, 15], and power [36, 39] that could potentially be used as modules.

3.3 Utility function

Using a utility function, administrators can communicate the cost and benefit structure of their environment to the provisioning tool. This allows the tool to make design trade-offs that increase the value of the system in their specific environment, “solving for” the most cost-effective level of each metric. The utility function is a mathematical expression that uses one or more of the system metrics to rank potential configurations. The function serves to collapse the many axes of interest to an administrator (the system metrics) into a single utility value that can be used by the provisioning tool.

While the function can be any arbitrary expression based on the system metrics, the purpose is to generate configurations that match well with the environment in which the system will be deployed. This argues for an approach that uses business costs and benefits as the basis for the utility function. Typically, it takes the form of a number of independent sub-expressions that are summed to form the final utility value. For example, an online retailer may derive a large fraction of its income from its transaction processing workload. Based on the average order amount, the fraction of transactions that are for new orders, and the average number of I/Os per transaction, the retailer may determine that, on average, its OLTP workload generates 0.1¢ per I/O. This would lead to an expression for the annualized revenue such as:

$$Revenue = \$0.001 \cdot IOPS_{WL} \cdot AV_{DS} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}} \right)$$

Here, the revenue is tied to the throughput of the workload (in I/Os per second). It is also scaled by the fractional availability of the dataset because revenue is not generated when the dataset is unavailable. Finally, it is converted to an annualized amount.

The administrator would also want to add expressions for the annualized cost of repair and lost productivity during downtime (e.g., \$10,000 per hour):

$$Cost_{downtime} = \left(\frac{\$10,000}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right)$$

The cost of losing the dataset (e.g., \$100 M) would be scaled by the annual failure rate:

$$Cost_{dataloss} = \$100 \text{ M} \cdot AFR_{DS}$$

These individual expressions of annual revenue and costs would be combined to form the utility function:

$$Utility = Revenue - Cost_{downtime} - Cost_{dataloss}$$

While this example has examined only a single workload and dataset, functions for other workloads and datasets in the same system can be included. In the case of independent applications (i.e., their associated revenue and costs are independent of other applications), such as combining an e-mail system with the e-commerce application, the components of utility from the different applications could be summed to produce a single composite function for the entire system. For environments where multiple applications depend on each other, a simple summation of the utility functions (i.e., assuming additive independence) may not be appropriate. Instead, the combination can be expressed by examining the benefits and costs of the combined service. For example, a web server and a database may be dependent, leading to an expression for the overall web application’s cost of downtime as a function of the availability of both the database and the web server (i.e., they must both be available to provide service).

The choice to use annualized amounts is arbitrary (e.g., hourly or monthly rates could be used as well), but all utility expressions need to share the same time frame to ensure they are scaled accurately relative to each other.

The benefit of taking a business-cost approach is that the utility function can be derived from an analysis of the business’ needs and objectives, reducing the need to invent a set of arbitrary requirements. While it still requires a thorough understanding of the application (e.g., to produce an estimate of the revenue per I/O), there is potential for the application, or its management tool, to assist. Future “utility-aware” applications could translate between their own high-level metrics and those of the storage system, again moving the level of specification closer to the administrator. For example, a database could provide a model for its expected transactions per second as a function of storage system performance metrics, allowing an administrator to express revenue as a function of the transactions per second achieved by the database.

3.4 Solver

The purpose of the solver is to generate improved storage system configurations based on the feedback provided

by the utility function. The solver is attempting to optimize a bin-packing problem wherein it must assign the datasets to storage nodes while attempting to maximize the administrator-provided utility function.

Developing an efficient solver can be difficult because the value (utility) of the storage system is only indirectly connected to the raw configuration settings (e.g., m , n , and l) the solver manipulates. The effect of the configuration on the utility value passes through both the system models and the utility function. Because this utility function is supplied by the system administrator at run-time, the tool designer cannot know how a configuration change is likely to affect utility, complicating the process of finding an efficient optimization algorithm.

4 A utility-based provisioning tool

We have implemented a utility-based provisioning tool that is targeted toward a cluster-based storage architecture [1] in which each client communicates directly to individual storage nodes. Datasets are spread across storage nodes using m -of- n data encodings, and the n data fragments may be declustered across an arbitrary set of l storage nodes. Workloads are statically assigned to a client, and they target a single dataset. However, clients may run multiple workloads, and datasets may be accessed by multiple workloads.

The system is described by the set of clients and workloads, the datasets they access, and the storage nodes that are available for use by the tool. Each component is described by a set of attributes specific to that component type. For example, storage nodes have attributes that describe their raw capacity, disk positioning time, streaming bandwidth, as well as network latency and bandwidth. Table 2 lists each component type and the attributes that are used to describe it.

A candidate configuration describes the mapping of each dataset onto storage nodes. The mapping is a tuple, $\{dataset, m, n, \text{list}\{storage\ nodes\}\}$, for each dataset in the system description, and the union of the storage node lists across these tuples determines the set of storage nodes in the configuration.

Our prototype tool is implemented in approximately 5000 lines of Perl. Text configuration files are used to define the characteristics of the system components and the utility function. The tool is designed to work with heterogeneous components, allowing each client, workload, storage node, and dataset to be unique.

The remainder of this section describes the tool's models and metrics, how utility is specified, and the implemen-

– Attributes of Components –

Client: <ul style="list-style-type: none"> • CPU time for data encode/decode (s) • Network streaming bandwidth (MB/s) • Network latency (s)
Dataset: <ul style="list-style-type: none"> • Size of the dataset (MB)
Storage node: <ul style="list-style-type: none"> • Annual failure rate (%) • Fractional availability of the node (%) • Disk capacity (MB) • Purchase cost (\$) • Max streaming bandwidth (MB/s) • Initial positioning time (s) • Network streaming bandwidth (MB/s) • Network latency (s) • Power consumption (W)
Workload: <ul style="list-style-type: none"> • Average request size (kB) • Multi-programming level for closed-loop workload • Think time for closed-loop workload (s) • Fraction of non-sequential I/Os (%) • Fraction of I/Os that are reads (%)

Table 2: Main components and their attributes – This table lists each of the main component types used in the system description for the provisioning tool. Listed with each of the component types is the set of attributes that define its properties. Each instance of a component (e.g., each storage node) may have different values for these attributes, allowing the tool to evaluate heterogeneous configurations.

tation of the solver. It concludes with an evaluation of the efficiency of the tool.

4.1 Models and metrics

The system models are plug-in modules that examine a configuration and produce metrics that describe the expected characteristics and level of service the system would provide. Additional models can be easily added to the existing list, expanding the menu of metrics that can be used by the utility function. The discussion below describes the existing set of models and the metrics that they provide. These models cover a wide range in their complexity and analysis techniques, highlighting the versatility of this architecture. While there exist potentially more accurate models for each of these metrics, the existing models provide sufficient detail to evaluate the value of utility for storage provisioning.

Performance: The performance model is the most intricate of the current models. It provides utilization

estimates for disk and network resources as well as throughput and latency estimates for the workloads. This information is derived from a closed-loop queueing model that is constructed and solved based on the system description and dataset assignments in the candidate configuration. The model is similar to that described by Thereska et al. [33], with queueing centers for the clients' CPU and network as well as the storage nodes' network and disk. The service demand placed on each queueing center is based on the encoding parameters for each dataset as well as the read/write ratio, I/O size, and sequentiality of the workloads. Workload intensity is specified as a multiprogramming level and a think time. These correspond to a maximum level of parallelism and an application's processing delay between I/Os, respectively. The open source PDQ [18] queueing model solver is used to analyze the model via Mean Value Analysis.

Availability: The availability model estimates the fractional availability of each dataset based on the availability of the storage nodes that the dataset is spread across and the encoding parameters used. The module calculates the dataset's availability as:

$$AV = \sum_{f=0}^{n-m} \binom{l}{l-f} A_{SN}^{l-f} (1 - A_{SN})^f$$

where n and m are the encoding parameters for the dataset, l is the number of storage nodes that the data is spread across, and A_{SN} is the minimum individual availability of the set of l storage nodes. The dataset is considered to be available as long as no more than $n - m$ of the l storage nodes are down.² The formula above sums the probability for each "available" state ($f = 0 \dots (n - m)$).

This model makes some simplifications, such as using a single availability value for all nodes. It also assumes independent failures of storage nodes, an assumption that has been called into question [28, 30] but is nonetheless sufficient for this study.

Reliability: The reliability model uses a Markov chain to estimate the annual failure rate for a dataset. The chain has $n - m + 2$ states, representing the number of device failures. The final state of the chain is an absorbing state, representing a data loss event. The transition rates for device failures are calculated as the number of storage nodes that contain fragments for the dataset (l) times the failure rate of the individual nodes. In the case where nodes have differing failure rates, the maximum is used, producing a conservative estimate. Repair operations are handled by re-encoding the dataset. The re-encode operation repairs all failures in the same operation, caus-

²We focus on a synchronous timing model and a crash failure model for the storage nodes.

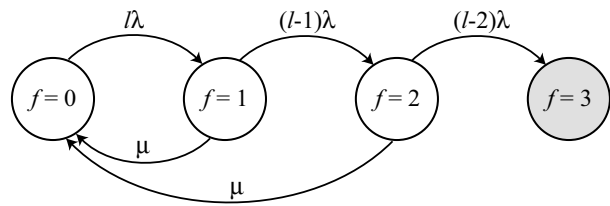


Figure 3: Markov chain for a 1-of-3 data encoding – A 1-of-3 data encoding is able to withstand up to 2 failures without losing data; the third failure ($f = 3$) results in a data loss. The transition rates between states are, in the case of failure transitions, related to the number of storage nodes that hold data for this dataset, l , and the annual failure rate of the nodes. The repair rate is calculated as a fixed fraction of the individual storage nodes streaming bandwidth (e.g., 5% of the slowest node) and the size of the dataset. A single repair operation is able to fix multiple failures simultaneously.

ing all repair transitions to lead to the failure-free state. The repair rate is based on the size of the dataset and a fixed fraction of the streaming bandwidth from the slowest storage node in the data distribution. The chain is solved for the expected time until the absorbing state is reached using the technique described by Pâris et al. [27]. Figure 3 shows an example chain for a 1-of-3 encoding. This reliability model represents only the likelihood of losing data from the primary storage system (i.e., it does not account for remote mirroring or external backup). More detailed models incorporating backup and remote replication, such as those by Keeton et al. [21], could be used.

Capacity: The capacity model calculates the storage blowup of each dataset due to the redundancy of the encoding scheme. The storage blowup is calculated as: $\frac{n}{m}$. It also calculates the capacity usage of each storage node based on the datasets they store.

Cost and power: The purchase cost and power models are very similar, producing system-wide metrics based on the quantity and type of storage nodes that are used. Each storage node type has fixed cost and power attributes. When a storage node is used in a configuration, it is assumed to cost a fixed amount to purchase and consume a fixed amount of power. These cost and power attributes are then summed to produce the system-wide metrics for cost and power consumption (e.g., a system with three storage nodes that cost \$10 k each would have a total system cost of \$30 k).

Table 3 lists the set of metrics provided by the current system models. This list provides the framework for creating utility functions to evaluate configurations. Additional models can be added to expand the menu of metrics available for the administrator to express his ob-

– System Metrics –

Client:
<ul style="list-style-type: none"> • CPU utilization (%) • Network utilization (%)
Dataset:
<ul style="list-style-type: none"> • Annual failure rate (%) • Fractional availability (%) • Capacity blowup from encoding • Mean time to failure (hr) • “Nines” of availability
Storage node:
<ul style="list-style-type: none"> • Raw capacity consumed (MB) • Capacity utilization (%) • Disk utilization (%) • Network utilization (%) • Power consumed (W)
System-wide:
<ul style="list-style-type: none"> • Total capacity consumed (MB) • Capacity utilization (%) • System power consumed (W) • Total system cost (\$)
Workload:
<ul style="list-style-type: none"> • Bandwidth (MB/s) • Throughput (IO/s) • Request latency (s)

Table 3: Storage metrics provided by system models – This table lists the metrics that are calculated for candidate configurations by the current set of system models. The table is organized by the component to which the metric refers. With the exception of the system-wide metrics, these metrics are calculated for each instance of a component (e.g., each storage node has a disk utilization that is calculated based on the accesses that it receives).

jectives. For example, an additional availability model could be created that estimates the frequency and duration of outages instead of just a fractional availability.

4.2 Specifying utility

The provisioning tool’s interface for specifying utility allows the use of an arbitrary function that assesses the metrics for the current system configuration. The utility function returns a single floating point number that is the utility value for the configuration. It is specified in the text configuration file as a block of Perl syntax that is **eval()**-ed when the configuration is loaded. This code has the ability to use any of the metrics listed in Table 3 when computing utility. Maintaining such a flexible interface to specify utility has proven valuable for experimentation. It allows not only utility functions based on business costs (as discussed in Section 3.3) but also util-

ity functions that implement strict priorities (e.g., first obtain 4 “nines” of availability, next achieve 300 IO/s, then minimize capacity utilization).

4.3 Solver

While many optimization techniques could be employed to generate candidate configurations, we have chosen to use a solver based on a genetic algorithm [26]. It refines a population of candidate solutions over a number of generations. Each generation contains a population of 100 candidates that are evaluated by the models and utility function. The utility value is used to generate a *fitness* value for each candidate. These fitness values are used to create a new population of candidates, and the process repeats. The creation of a new population based on the existing population occurs via *selection*, *crossover*, and *mutation* operations. These operations each introduce randomness into the search, attempting to avoid local maxima and maintain diversity within the population. The solver continues, producing configurations with higher utility, until some stopping condition is reached. As the solver progresses through a number of generations, the observed gains in utility diminish. The solver terminates if there has been no improvement upon the best configuration for 40 generations.

Fitness: The fitness value determines how likely a candidate is to be selected for reproduction into the next generation. The fitness value accounts for both the utility of the candidate as well as the feasibility of the solution. Due to capacity constraints, not all configurations are feasible. To bias the solution toward feasible, high-utility solutions, fitness is calculated as:

$$fitness = \begin{cases} utility & \text{if } (utility \geq 2 \ \& \ OCC = 0) \\ \frac{1}{1+OCC} & \text{if } (OCC > 0) \\ \frac{1}{3-utility} + 1 & \text{otherwise} \end{cases}$$

OCC is the sum of the over-committed capacity from the individual storage nodes (in MB). For those infeasible solutions, the fitness value will be between zero and one. When the solution is feasible, *OCC* = 0 (no storage nodes are over-committed). Utility values less than two are compressed into the range between two and one, eliminating negative utility to be compatible with the Roulette selection algorithm and ensuring that all feasible solutions have a fitness value greater than the infeasible ones. This type of penalty scheme for infeasible solutions is similar to that used by Feltl and Raidl [13].

Selection function: Using the fitness value for guidance, a selection function probabilistically chooses candidates to use as a basis for the next generation of solutions. The solver can use either Tournament or Roulette selection

algorithms [26], but it defaults to Tournament. Empirically, the solver performs better using Tournament selection for utility functions that are ordinal (where the utility value specifies only an ordering of candidates, not “how much” better a given solution is). With Tournament selection, two candidates are chosen at random (uniformly) from the current population. From these two candidates, the one with the larger fitness value is output as the selected candidate. The result of this process is that the selection algorithm chooses candidates weighted by their rank within the current population.

Roulette selection, on the other hand, chooses candidates for the next generation with a probability proportional to the relative magnitude of their fitness values. For example, a candidate with a fitness value of 100 would be twice as likely to be selected as one with a value of 50.

Candidate representation: For use in the genetic algorithm (GA), the storage configuration space is encoded as a matrix. This matrix has one row for each dataset and one column for each storage node. Each location in the matrix may take on integer values between zero and three, inclusive. Any non-zero value at a particular location is used to indicate that the dataset (represented by the row) is stored on the storage node (represented by the column). The values of the m and n encoding parameters for a dataset are determined by the number of entries in that dataset’s row with values greater than two and one, respectively. For example, a row of $[0\ 3\ 2\ 1\ 2]$ denotes an encoding of 1-of-3, with fragments stored on nodes two, three, four, and five ($l = 4$). This matrix representation was chosen because of the relative ease of maintaining the invariant: $1 \leq m \leq n \leq l$. This representation ensures the relationship of m , n , and l . The only condition that must be verified and corrected is that m must be at least one. That is, there must be at least one “3” in each row of the matrix for the encoding to be valid.

When the GA is initialized (i.e., the first generation is created), the matrix is generated randomly, with values from zero to three in each cell. These values are then changed via the crossover and mutation operations as the optimization progresses.

Crossover operator: The crossover operation combines two candidates from the current generation to produce two new candidates for the next generation. The intuition behind this step is to create new solutions that have properties from both of the original candidates (potentially achieving the best of both). The solver uses *uniform crossover* to achieve this mixing. In uniform crossover, each “gene” has an independent 50% probability of coming from either original candidate. Crossover is performed at the “dataset” level — an entire row of the configuration matrix is selected as a single unit. This ap-

proach was chosen because the individual values within a row have little meaning when taken independently, and combining at the dataset level is more likely to form a new set of candidates with properties of the original two.

Mutation operator: The mutation operator is used to add randomness to the search. It operates on a single candidate at a time by changing a random location in the configuration matrix to a new random integer in the range of zero to three. Before a particular value is changed, it is verified that the new value will not cause the configuration to be invalid — a location with a current value of “3” can only be changed if there is at least one other “3” in that row. If a conflict is found, a different location is chosen for mutation.

The GA also supports being used for “incremental” provisioning, wherein additional datasets, and potentially hardware, are added to an existing system. In this case, the existing datasets can be held constant by omitting them from the matrix representation, only using them when utility is calculated. Evaluating whether an existing dataset should be reassigned as a part of the incremental provisioning process is left as an area of future work.

4.4 Tool effectiveness

For utility to be useful at guiding the provisioning of storage, it must be possible to reliably find configurations that have high (near optimal) utility. In the following experiments, we show that the solutions produced by the solver approach the optimum and that the solver is able to quickly find good solutions for difficult problems. Although better algorithms likely exist, these experiments demonstrate that it is possible to create an effective utility-based provisioning tool.

4.4.1 Convergence toward the optimum

This experiment compares the solutions produced by the genetic solver with the optimal solution produced by an exhaustive search of the configuration space. To constrain the problem so that it can be solved by exhaustive search, configurations may use a maximum of eight storage nodes.

The utility function used for this scenario is identical to the example presented in Section 3.3 but presented as

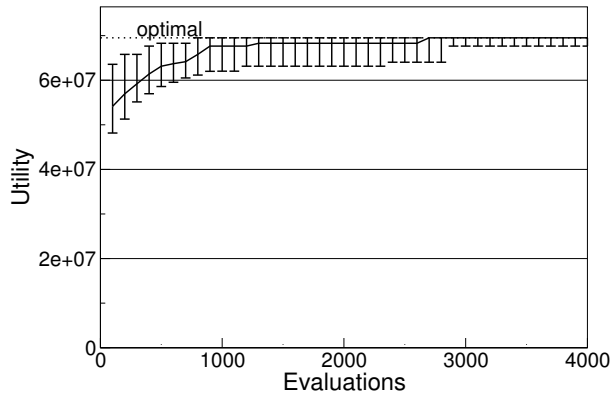


Figure 4: Convergence of the genetic solver – This graph shows how quickly the solver converges toward the optimal storage configuration. The line is the median over 100 trials, and the error bars indicate the 5th and 95th percentiles.

utility (costs are represented as negative utility):

$$\begin{aligned}
 \text{Utility} &= U_{\text{revenue}} + U_{\text{dataloss}} + U_{\text{downtime}} \\
 U_{\text{revenue}} &= \$0.001 \cdot AV_{DS} \cdot IOPS_{WL} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}} \right) \\
 U_{\text{dataloss}} &= -\$100 \text{ M} \cdot AFR_{DS} \\
 U_{\text{downtime}} &= \left(\frac{-\$10,000}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right)
 \end{aligned}$$

For this experiment, there are two identical clients, each with one workload, issuing I/O requests to separate datasets. The total utility is the sum across both of the workloads and datasets. The simulated storage nodes have 5.5 ms average latency and 70 MB/s max streaming bandwidth from their disk. They are assumed to have an individual availability of 0.95 and an annual failure rate of 1.5%.

Figure 4 shows the results for this experiment. The exhaustive solution produces a utility of 6.95×10^7 , using an encoding of 1-of-2 declustered across 4 storage nodes, and the two datasets are segregated onto their own set of 4 storage nodes. This optimal utility is shown as the dotted line near the top of the graph. The GA solver approaches this value quickly. Within five generations (500 total configurations evaluated), the median of 100 trials is within 10% of the optimum, and the bottom 5% achieves this level after just twelve generations (1200 total evaluations). Allowing for equivalent configurations, 3336 out of 7.9×10^6 total configurations are within 10% of the optimum. The utility values across the configuration space range from -7.23×10^7 to 6.95×10^7 .

4.4.2 Finding rare solutions

The previous experiment provided an example showing that the solver quickly approaches the optimal solution. This experiment will explore how well the solver is able to find rare solutions. The ability to find rare solutions is important because some combinations of workloads, hardware, and utility functions have the characteristic that there are very few configurations within a small percentage of the optimal solution.

For this experiment, the same storage nodes are used, but the number of clients, datasets, and workloads are scaled together with a 1:1:1 ratio to control the problem’s difficulty. A (contrived) utility function is constructed so that it is possible to predict the number of “desirable” configurations as a fraction of the total number of possible storage configurations. For this experiment, the definition of “desirable” is that all datasets should have at least 4 “nines” of availability. Availability is used for this experiment because the data distribution is the sole determinant of its value, and one dataset’s distribution does not affect another’s availability. Performing an exhaustive search with a single dataset, 464 of 2816 or 16.5% of the possible distributions meet the 4 nines criteria. By scaling the number of datasets in the scenario, solutions where all datasets have 4 nines of availability can be made an arbitrarily small fraction of the possible configurations. For example, with three datasets, $\left(\frac{464}{2816}\right)^3 = .4\%$ of the possible configurations have 4 nines for all three workloads.

The utility function for this scenario is:

$$U = \frac{1}{S} \cdot \sum_{i=1}^S \min(NINES_{DS_i}, 4)$$

S is the scale factor, corresponding to the number of datasets. This utility function will achieve its maximum value, four, when all datasets achieve at least 4 nines of availability. To ensure all possible data distributions are valid as the number of datasets are scaled, the size of the datasets relative to the storage nodes’ capacity is chosen to ensure the system is not capacity constrained.

Figure 5 shows how the solver performs as the number of datasets is scaled from 5 up to 50. The graph plots the difficulty (the reciprocal of the fraction of configurations with 4 nines) of finding a 4 nines solution versus the number of configurations the solver evaluates before finding the first. It can be seen that exponential increases in the rarity of “desirable” solutions result in an approximately linear growth in the number of configurations that must be evaluated by the solver.

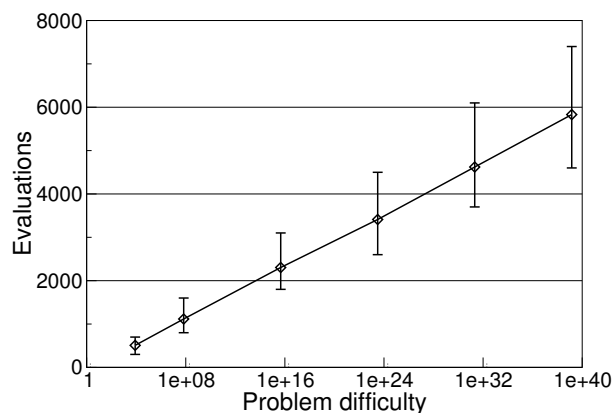


Figure 5: Scalability of genetic solver – This graph shows how the solution time changes as a function of how difficult the problem is to solve. The x-axis is the inverse probability that a random configuration is a valid solution ($\frac{1}{\text{Pr}[\text{valid_solution}]}$). The graph shows the mean number (across 100 trials) of configurations that are evaluated before finding a valid solution. The error bars indicate the 5th and 95th percentiles.

4.4.3 Solution speed

Our provisioning tool, implemented in Perl, solves the above “4 nines” scenarios in a few minutes. These measurements were taken on a Dell PowerEdge 1855 with dual 3.40 GHz Intel Xeon CPUs and 3 GB of RAM, running Linux kernel version 2.2.16 and Perl 5.8.8. The provisioning tool used only one of the two CPUs.

Figure 6 shows the average time required to evaluate one generation of 100 candidates for each of the problem sizes. It divides this per-generation time into three categories, corresponding to the three main system components (models, utility function, and genetic solver). The majority of the runtime (and growth in runtime) is consumed by the system models, suggesting that efficient (but still accurate) models are a key to effective utility-based provisioning. The time required to calculate the utility value from the system metrics is too small to be visible on the graph.

5 Case studies

The benefits of using utility as the basis for storage provisioning can be seen by examining several case studies. This section uses our utility-based provisioning tool to explore three different scenarios, highlighting several important benefits of using utility to evaluate designs.

The simulated system components used for the case studies are described in Table 4. The same workload descrip-

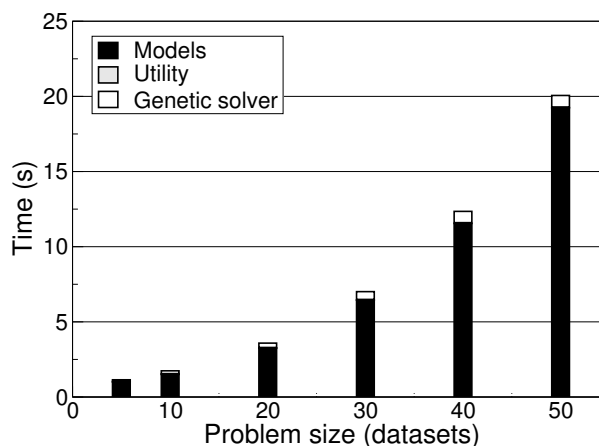


Figure 6: Speed of genetic solver – This graph shows how the evaluation time changes as a function of the problem size. The x-axis is the number of datasets being optimized. There is one client and workload for each. The y-axis is the time required to fully evaluate one generation of candidate solutions (100 configurations). The system models consume the majority of the processing time, and the actual evaluation of the utility function is insignificant.

tion is used for the first two case studies, however the third uses a workload description more appropriate to the described application (trace processing).

5.1 Performance vs. availability

Provisioning and configuration decisions affect multiple system metrics simultaneously; nearly every choice involves a trade-off. By using utility, it is possible to take a holistic view of the problem and make cost-effective choices.

Conventional wisdom suggests that “more important” datasets and workloads should be more available and reliable and that the associated storage system is likely to cost more to purchase and run. To evaluate this hypothesis, two scenarios were compared. Each scenario involved two identical workloads with corresponding datasets.

The utility functions used for the two scenarios are similar to the previous examples, having a penalty of \$10,000 per hour of downtime and \$100 M penalty for data loss. In this example, the cost of electricity to power the system was also added at a cost of \$0.12 per kWh, and the purchase cost of the system (storage nodes) was amor-

Client			
CPU	0.2 ms	Net latency	125 μ s
Net bw	119 MB/s		
Dataset			
Size	100 GB		
Storage node			
AFR	0.015	Avail	0.95
Capacity	500 GB	Cost	\$5000
Disk bw	70 MB/s	Disk latency	5.5 ms
Net bw	119 MB/s	Net latency	125 μ s
Power	50 W		
Workload (§5.1, §5.2)			
I/O size	8 kB	MP level	5
Think time	1 ms	Rand frac	0.5
Read frac	0.5		
Workload (§5.3)			
I/O size	32 kB	MP level	10
Think time	1 ms	Rand frac	0.0
Read frac	1.0		

Table 4: Components used for case studies – This table lists the main attributes of the components that are used as a basis for the case study examples. The client is based on measurements from a 2.66 GHz Pentium 4. The storage node data is based on the data sheet specifications for a single disk drive, combined with a 1 Gb/s network connection, processing and cache.

tized over a three year expected lifetime:

$$\begin{aligned}
Utility &= U_{perf} + U_{avail} + U_{rel} + U_{power} + U_{cost} \\
U_{perf} &= (\text{see below}) \\
U_{avail} &= \left(\frac{-\$10,000}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right) \\
U_{rel} &= -\$100 \text{ M} \cdot AFR_{DS} \\
U_{power} &= \left(\frac{-\$0.12}{\text{kW} \cdot \text{hr}} \right) \cdot Power \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right) \cdot \left(\frac{1 \text{ kW}}{1000 \text{ W}} \right) \\
U_{cost} &= \left(\frac{-Cost}{3 \text{ yr}} \right)
\end{aligned}$$

The two scenarios differed only in the revenue they generated. The first generated 0.1¢ per I/O while the second only generated 0.01¢:

$$\begin{aligned}
U_{perf0.1} &= \$0.001 \cdot AV_{DS} \cdot IOPS_{WL} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}} \right) \\
U_{perf0.01} &= \$0.0001 \cdot AV_{DS} \cdot IOPS_{WL} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}} \right)
\end{aligned}$$

Based on this revenue difference, it would be easy to assume that the workload generating more revenue is “more important” than the other, requiring a higher (or

at least the same) level of data protection. This assumption fails to account for the compromises necessary to achieve a particular level of availability.

Table 5 shows the results of provisioning these two systems. It shows both the metrics and costs for each part of the utility function. The table shows the optimal configuration for each scenario: 1-of-2 declustered across 6 (1/2/6) for the 0.1¢ scenario and 1-of-3 across 7 (1/3/7) for the 0.01¢ scenario. As a point of comparison, it also shows the results of using the other scenario’s optimal configuration for each.

Examining the various contributions to the total utility, it can be seen that the main trade-off between these two scenarios is in performance versus availability. For the scenario with the higher revenue per I/O, it is advantageous to choose the data distribution with higher performance at the cost of lower availability (1/2/6), because the revenue generated by the extra 209 I/Os per second per workload more than offsets the cost incurred by the extra downtime of this configuration. For the lower revenue scenario, the extra throughput cannot offset the availability difference, causing the lower performing, more available data distribution (1/3/7) to be preferred.

It is important to remember that these configurations are a balance of competing factors (mainly performance and availability in this case). Taking this trade-off to an extreme, such as choosing a very high performance data distribution with no regard to availability and reliability, results in poor utility. For example, using a 1/1/6 distribution for the 0.1¢ scenario provides only \$33.7 M in utility because the reliability and availability costs now dominate.

Sacrificing availability for performance in a business scenario goes against the conventional wisdom of storage provisioning. However, by using utility to analyze potential configurations, the multiple competing objectives can be examined analytically, providing evidence to explain and justify a particular storage solution.

5.2 Storage on a budget

Even with the ability to quantify the costs and benefits of a particular storage solution, it is not always possible for a system administrator to acquire the optimal system due to external constraints. For example, the “optimal” storage system for a particular scenario may be too expensive for the system administrator to purchase with his limited budget. Presenting the administrator with this solution does him no good if he cannot afford it. Using utility, he has the ability to scale down this solution to find the best option that fits within his budget.

Using the previous 0.01¢ scenario as an example, the optimal solution uses fourteen storage nodes (seven for each dataset) and costs \$70 k. For an administrator whose budget cannot accommodate this purchase, this solution is unworkable. Table 6 compares this optimal solution to two alternatives that have constraints on the total cost of the storage hardware. The first alternative limits the total hardware budget to \$30 k, and the second further reduces it to \$20 k. Notice that these two alternatives use six and four storage nodes respectively (at \$5000 each) to stay within their budget. The “AP cost” in the table reflects this cost amortized over the system’s expected three year lifetime.

With each reduction in budget, the total system utility decreases as expected, but the chosen configuration at each level still balances the relevant system metrics to maximize utility as much as possible. The reduction from the optimum (\$70 k) to \$30 k results in choosing a configuration that sacrifices some availability to gain performance, resulting in only a 2% loss of overall utility. The reduction to \$20 k from the optimum leads to a 10% loss of utility, as performance is significantly impacted.

As this example illustrates, utility presents the opportunity to make trade-offs even among non-optimal or in less than ideal situations. This ability to account for external constraints makes utility-based tools more helpful than those that perform only minimum cost provisioning by allowing solutions to be identified that conform to real-world constraints.

5.3 Price sensitivity

Even without budgetary constraints, the price of the storage hardware can impact the proper solution. Consider the case of an academic research group whose students process file system traces as a part of their daily work. The trace processing application reads a trace (27 GB on

Total budget	\$70 k	\$30 k	\$20 k
Distribution	1/3/7	1/2/3	1/2/2
Performance	\$6.5 M	\$6.7 M	\$5.8 M
Availability	−\$329 k	−\$636 k	−\$219 k
Reliability	−\$0.02	−\$163	−\$81
Power	−\$736	−\$316	−\$210
AP cost	−\$23 k	−\$10 k	−\$6.6 k
Total utility	\$6.2 M	\$6.1 M	\$5.6 M

Table 6: Utility can be used to design for limited storage budgets – The optimal system configuration costs a total of \$70 k, giving an amortized purchase (AP) cost of \$23 k, but the utility function can be used to choose the best configuration that fits within other (arbitrary) budgets as well. Limiting the budget constrains the total hardware available, and the utility function guides the solution to make cost effective trade-offs as the system capabilities are scaled down to meet the limited budget.

average in this scenario) sequentially and generates a set of summary statistics. Assuming that the students cost \$35 per hour³, that they wait for the results of a run, and that there are 250 runs per year (approximately one for each regular workday), the total cost incurred is:

$$U_{perf} = \left(\frac{-\$35}{\text{hr}} \right) \cdot \left(\frac{27 \text{ GB}}{\text{run}} \right) \cdot \frac{1}{BW_{WL}} \cdot \left(\frac{250 \text{ runs}}{\text{yr}} \right) \cdot \left(\frac{\text{hr}}{3600 \text{ s}} \right) \cdot \left(\frac{1024 \text{ MB}}{\text{GB}} \right) = \frac{67200}{BW_{WL}}$$

If the traces are lost from primary storage, it is projected to require 15 hours of administrator time (at \$35 per hour) to re-acquire and restore the traces:

$$U_{rel} = \left(\frac{-\$35}{\text{hr}} \right) \cdot (15 \text{ hr}) \cdot AFR_{DS}$$

³This amount is the cost to the research program, not what the students are paid, unfortunately.

Distribution	Metric values		0.1¢ per I/O		0.01¢ per I/O	
	1/2/6	1/3/7	1/2/6 (opt)	1/3/7	1/2/6	1/3/7 (opt)
Performance	1250 IO/s	1041 IO/s	\$76.3 M	\$65.4 M	\$7.6 M	\$6.5 M
Availability	1.5 nines	2.4 nines	−\$2.8 M	−\$329 k	−\$2.8 M	−\$329 k
Reliability	2.0×10^{-6} afr	1.1×10^{-10} afr	−\$407	−\$0.02	−\$407	−\$0.02
Power	600 W	700 W	−\$631	−\$756	−\$631	−\$756
Purchase cost	\$60,000	\$70,000	−\$20 k	−\$23 k	−\$20 k	−\$23 k
Total utility			\$73.4 M	\$65.1 M	\$4.7 M	\$6.2 M

Table 5: Effect of workload importance on provisioning decisions – A workload that generates more revenue per I/O should not necessarily have a higher level of data protection. This table compares two scenarios that differ only in the average revenue generated per completed I/O. The “more valuable” dataset’s optimal data distribution is less available than that of the “less valuable” dataset because the cost of the additional downtime is more than offset by the additional performance of a less available data distribution. The data distributions in the table are written as: *m/n/l*.

	Expensive (opt) 1/2/2	Cheap (same) 1/2/2	Cheap (opt) 1/3/3
Performance	−\$874	−\$874	−\$862
Availability	−\$1534	−\$1534	−\$77
Reliability	−\$0	−\$0	−\$0
Power	−\$105	−\$105	−\$158
AP cost	−\$6667	−\$1333	−\$2000
Total utility	−\$9180/yr	−\$3846/yr	−\$3097/yr

Table 7: The price of the storage hardware affects the optimal storage configuration – The optimal configuration using “expensive” (\$10 k each) storage nodes is two-way mirroring. However, if the cost of the storage nodes is reduced to \$2000, it is now advantageous to maintain an additional replica of the data to increase availability. This additional storage node results in an almost 20% decrease in expected annual costs for the storage system.

If the traces are unavailable, the administrator and one student will be occupied troubleshooting and fixing the problem:

$$U_{avail} = \left(\frac{-\$70}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right)$$

The storage system must be powered, and the purchase cost will be spread across a three year lifetime:

$$U_{power} = \left(\frac{-\$0.12}{\text{kW} \cdot \text{hr}} \right) \cdot \text{Power} \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right) \cdot \left(\frac{1 \text{ kW}}{1000 \text{ W}} \right)$$

$$U_{cost} = \left(\frac{-\text{Cost}}{3 \text{ yr}} \right)$$

Provisioning this system using storage nodes as described in Table 4 but using a total cost of \$10 k per node (\$3.3 k annualized) leads to a solution using two storage nodes and 2-way mirroring. The first column of Table 7 shows a breakdown of the costs using these “expensive” storage nodes. If the cost of a storage node is reduced to \$2 k each (\$667 annualized), the total costs obviously decrease due to the lower acquisition cost (second column of Table 7). More interestingly, the optimal configuration for the system also changes because it is now cost effective to purchase an additional storage node. By comparing the last two columns in the table, the additional annualized cost of the third storage node (\$667) is more than offset by the increase in availability that it can contribute (\$1457). In fact, this new configuration provides almost a 20% reduction in annual costs.

6 Conclusion

Producing cost-effective storage solutions requires balancing the costs of providing storage with the benefits that the system will provide. Choosing a proper storage configuration requires balancing many competing system metrics in the context where the system will be deployed. Using utility, it is possible to bring these costs and benefits into the same framework, allowing an automated tool to identify cost-effective solutions that meet identified constraints.

This paper illustrates the value of utility-based provisioning with three case studies. The first case study shows an example where trade-offs exist between metrics, such as performance and availability, and utility provides a method to navigate them. The second shows that utility functions are flexible enough to be used in the presence of external constraints, such as a limited budget. The third shows that provisioning a storage system is not just limited to finding the “minimum cost” solution that meets a set of requirements, because the system cost can have an impact on the solution.

This investigation shows the potential for using utility to guide static storage provisioning. By analyzing utility over time, there is also the potential to provide guidance for automated, online tuning as well. For example, a modified version of our provisioning tool could be used to generate new candidate configurations, evaluate long-term expected utility, and decide whether the change is advantageous and how fast to migrate or re-encode the data. Exploring such on-line use of utility is an interesting area for continuing work.

7 Acknowledgements

We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, Network Appliance, Oracle, Seagate, and Symantec) for their interest, insights, feedback, and support. Experiments were enabled by hardware donations from Intel, APC, and Network Appliance. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, and by the Army Research Office, under agreement number DAAD19-02-1-0389. Experiments were run using the Condor [24] workload management system.

References

- [1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile cluster-based storage. In *Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2005. ISBN 978-1-931971-39-3.
- [2] G. A. Alvarez, J. Wilkes, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, and A. Veitch. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001. ISSN 0734-2071. doi: 10.1145/502912.502915.
- [3] K. Amiri and J. Wilkes. Automatic design of storage systems to meet availability requirements. Technical Report HPL-SSP-96-17, Hewlett-Packard Laboratories, August 1996.
- [4] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, Hewlett-Packard Laboratories, July 2001.
- [5] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: Quickly finding near-optimal storage system designs. Technical Report HPL-SSP-2001-05, Hewlett-Packard Laboratories, 2001.
- [6] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Conference on File and Storage Technologies*, pages 175–188. USENIX Association, 2002. ISBN 978-1-880446-03-4.
- [7] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, November 2005. ISSN 0734-2071. doi: 10.1145/1113574.1113575.
- [8] A. AuYoung, L. Grit, J. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *International Symposium on High-Performance Distributed Computing*, pages 119–131. IEEE, 2006. ISBN 978-1-4244-0307-3. doi: 10.1109/HPDC.2006.1652143.
- [9] I. D. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *Symposium on Discrete Algorithms*, pages 661–670. Society for Industrial and Applied Mathematics, 2001. ISBN 978-0-89871-490-6.
- [10] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *International Workshop on Quality of Service*, pages 203–207. IFIP, 1997. ISBN 978-0-412-80940-8.
- [11] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *Symposium on Fault-Tolerant Computer Systems*, pages 432–441. IEEE, 1993. ISBN 978-0-8186-3680-6. doi: 10.1109/FTCS.1993.627346.
- [12] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, June 1982. ISSN 0360-0300. doi: 10.1145/356876.356883.
- [13] H. Feltl and G. R. Raidl. An improved hybrid genetic algorithm for the generalized assignment problem. In *Symposium on Applied Computing*, pages 990–995. ACM Press, 2004. ISBN 978-1-58113-812-2. doi: 10.1145/967900.968102.
- [14] S. Gaonkar, K. Keeton, A. Merchant, and W. H. Sanders. Designing dependable storage solutions for shared application environments. In *International Conference on Dependable Systems and Networks*, pages 371–382. IEEE Computer Society, 2006. ISBN 978-0-7695-2607-2. doi: 10.1109/DSN.2006.27.
- [15] R. Geist and K. Trivedi. An analytic treatment of the reliability and performance of mirrored disk subsystems. In *Symposium on Fault-Tolerant Computer Systems*, pages 442–450. IEEE, 1993. ISBN 978-0-8186-3680-6. doi: 10.1109/FTCS.1993.627347.
- [16] J. P. Gelb. System-managed storage. *IBM Systems Journal*, 28(1):77–103, 1989. ISSN 0018-8670.
- [17] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks*, pages 135–144. IEEE Computer Society, 2004. ISBN 978-0-7695-2052-0. doi: 10.1109/DSN.2004.1311884.
- [18] N. Gunther and P. Harding. PDQ (pretty damn quick) version 4.2. <http://www.perfdynamics.com/Tools/PDQ.html>, 2007.
- [19] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *International Symposium on High-Performance Distributed Computing*, pages 160–169. IEEE, 2004. ISBN 978-0-7695-2175-6. doi: 10.1109/HPDC.2004.1323519.
- [20] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993. ISBN 978-0-521-43883-4.
- [21] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2004. ISBN 978-1-931971-19-5.
- [22] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the road to recovery: Restoring data after disasters. In *European Systems Conference*, pages 235–248. ACM Press, 2006. doi: 10.1145/1217935.1217958.

- [23] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *International Workshop on Policies for Distributed Systems and Networks*, pages 3–12. IEEE, 2004. ISBN 978-0-7695-2141-1. doi: 10.1109/POLICY.2004.1309145.
- [24] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988. ISBN 978-0-8186-0865-0. doi: 10.1109/DCS.1988.12507.
- [25] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Conference on Measurement and Modeling of Computer Systems*, pages 37–48. ACM Press, 2007. doi: 10.1145/1254882.1254887.
- [26] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.
- [27] J.-F. Pâris, T. J. E. Schwarz, and D. D. E. Long. Evaluating the reliability of storage systems. Technical Report UH-CS-06-08, Department of Computer Science, University of Houston, June 2006.
- [28] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Conference on File and Storage Technologies*, pages 17–28. USENIX Association, 2007. ISBN 978-1-931971-50-8.
- [29] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Architectural Support for Programming Languages and Operating Systems*, pages 48–58. ACM Press, 2004. doi: 10.1145/1024393.1024400.
- [30] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Conference on File and Storage Technologies*, pages 1–16. USENIX Association, 2007. ISBN 978-1-931971-50-8.
- [31] E. St.Pierre. ILM: Tiered services & the need for classification. Technical tutorial, Storage Networking Industry Association, April 2007. http://www.snia.org/education/tutorials/2007/spring/data-management/ILM-Tiered_Services.pdf.
- [32] X. Tang and J. Xu. On replica placement for QoS-aware content distribution. In *Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 806–815. IEEE, 2004. ISBN 978-0-7803-8355-5.
- [33] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. In *International Conference on Autonomic Computing*, pages 187–198. IEEE, 2006. ISBN 978-1-4244-0175-8.
- [34] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):559–574, June 2004. ISSN 1045-9219. doi: 10.1109/TPDS.2004.9.
- [35] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *International Conference on Autonomic Computing*, pages 70–77. IEEE, 2004. ISBN 978-0-7695-2114-5. doi: 10.1109/ICAC.2004.1301349.
- [36] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning. PARAID: A gear-shifting power-aware RAID. *ACM Transactions on Storage*, 3(3):13, October 2007. ISSN 1553-3077. doi: 10.1145/1289720.1289721.
- [37] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *International Workshop on Quality of Service*, pages 75–91. IFIP, 2001. ISBN 978-3-540-42217-4.
- [38] J. Wolf. The Placement Optimization Program: A practical solution to the disk file assignment problem. *Performance Evaluation Review*, 17(1):1–9, May 1989. ISSN 0163-5999. doi: 10.1145/75108.75373.
- [39] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Symposium on Operating System Principles*, pages 177–190. ACM Press, 2005. ISBN 978-1-59593-079-8. doi: 10.1145/1095810.1095828.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login;*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see <http://www.usenix.org/membership/specialdisc.html>

For more information about membership, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patrons

Google
Microsoft Research
NetApp

USENIX & SAGE Partners

Ajava Systems, Inc.
DigiCert® SSL Certification
Raytheon
rTIN Aps
Splunk
Taos
Tellme Networks
Zenoss

USENIX Partners

Cambridge Computer Services, Inc.
cPacket Networks
EAGLE Software, Inc.
GroundWork Open Source Solutions
Hewlett-Packard
Hyperic
IBM
Infosys
Intel
Interhack
Oracle

Ripe NCC

Sendmail, Inc.
Sun Microsystems, Inc.
UUNET Technologies, Inc.
VMware

SAGE Partners

FOTO SEARCH Stock Footage and Stock Photography
MSB Associates

ISBN-13: 978-1-931971-56-0
ISBN-10: 1-931971-56-0



9 781931 971560